清源

程序设计

OpenGL

乔林　费广正　林杜
芦丽丹　　　马晓峰　编著

清华大学出版社
http://www.tup.tsinghua.edu.cn

内附光盘

OpenGL 程序设计

乔林　费广正等　编著

清 华 大 学 出 版 社

（京）新登字 158 号

内 容 简 介

OpenGL 是目前应用最为广泛的开放式图形编程标准，用户可以利用它编写出高质量的图形绘制程序。

本书从编写 OpenGL 程序的基础知识和基本概念出发，讲解了如何在 Win32 环境下正确设置环境以进行 OpenGL 程序设计；然后介绍了 OpenGL 的基本内容，如何在程序中读取三维图形数据，以及 OpenGL 的高级编程方法，并编写了一个屏幕保护程序和一个虚拟建筑物漫游程序。最后，重点介绍了 Windows 环境下对 OpenGL 功能的扩展及程序改进。

本书示例丰富、实用，语言通俗易懂，介绍全面，无论对大专院校的大学生、研究生，还是对科研院所的研究人员，都是一本三维图形技术难得的指导和参考书。此外，本书亦可作为从事计算机图形工作一般人员的三维图形技术引导，也非常适合作为 OpenGL 三维图形编程的培训教程。

版权所有，翻印必究。
本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

书　　名：OpenGL 程序设计
作　　者：乔林、费广正等
出 版 者：清华大学出版社（北京清华大学学研楼，邮编 100084）
　　　　　http://www.tup.tsinghua.edu.cn
责任编辑：胡伟卷
印 刷 者：北京市人民文学印刷厂
发 行 者：新华书店总店北京发行所
开　　本：787×1092　1/16　印张：29.5　字数：714 千字
版　　次：2000 年 4 月第 1 版　2000 年 4 月第 1 次印刷
书　　号：ISBN 7-900625-15-1
印　　数：0001~5000
定　　价：54.00 元

前言

目前，计算机图形学已进入三维时代，三维图形在人们周围无所不在。魅力无穷的计算机广告，扣人心弦的三维游戏，声势浩大、震撼人心的影视特效，无一不是三维图形大显身手的得意之作。

科学可视化、计算机动画和虚拟现实已成为近年来计算机图形学的三大热门话题，而这三大热门话题的技术核心均为三维图形。当前，三维图形在军事、航空、航天、医学、地质勘探、文化娱乐和艺术造型等方面有着十分广泛的应用。

众多的计算机工作者，尤其是广大的计算机爱好者，对计算机图形、图像怀着浓厚的兴趣。许多计算机爱好者在图形学这片土地上苦苦耕耘，创作出许多优秀的软件作品。他们在享受由此带来的乐趣的同时，也真真切切地感受到其中的艰辛。他们盼望有一种简单易用的编程标准，正像许多人当年盼望 Windows 界面编程标准一样，因为他们深知：只有标准的才是有生命力的。

1992 年，OpenGL 正式成为适用于各种计算机环境下的三维应用程序接口（3D API）。目前，它已成为国际上通用的开放式三维图形标准。Compaq，IBM，Intel，Microsoft，DEC，HP 和 SUN 等在计算机界具有主导地位的公司纷纷采用 OpenGL 图形标准。各种支持 OpenGL 的图形卡，3D 加速卡层出不穷，已成为计算机界继 CPU、内存、硬盘之后的又一快速增长点。逼真的 3D 游戏让人目不暇接，令人身临其境的虚拟现实也正向人们走来。这一切都必然吸引着更多的人加入到图形这一行列中，然而，国内介绍 OpenGL 的书可谓少而又少，致使不少人仍在苦苦地寻觅，焦急地等待。

事实上 OpenGL 是一种与硬件、窗口系统和操作系统相独立的一系列 API，它可以被集成到 Unix，Windows 95，Windows NT 和 X Window 等窗口系统中（这里尤其值得指出的是：作为软件界全球霸主的 Microsoft 为表示其支持 OpenGL 的决心，已将 OpenGL 作为核心模块之一加入到 Windows NT 3.5 以后的版本中）。OpenGL 由大量功能强大的图形函数组成，开发人员可以利用这些函数对整个三维图形进行光色渲染，从而方便地绘制出客观世界逼真的三维景象。

本书作者结合自己的实际开发经验，以及使用时的大量心得，通过丰富的教学实例，系统地讲述了 OpenGL 三维图形设计的基本概念和方法。

本书共由 20 章组成，第 1~第 4 章从编写 OpenGL 程序的基础知识和基本概念出发，讲解了如何在 Win32 环境下正确设置环境以进行 OpenGL 程序设计；第 5~第 8 章介绍了 OpenGL 的基本内容，包括几何图元的构造、坐标变换、颜色和光照等；第 9 章介绍了如何在程序中读取三维图形数据；第 10~第 16 章讲解了 OpenGL 的高级编程方法，内容包括：位图、字体和图像、纹理映射、各种特殊效果、帧缓存技术与动画、显示列表、参数曲面和曲线以及利用选择与反馈进行交互式图形程序设计；第 17 和第 18 章编写了两个实

际应用程序，一个屏幕保护程序和一个虚拟房间漫游程序。第 19、20 章重点讲解了 Windows 环境下对 OpenGL 功能的扩展及程序改进。

本书由乔林、费广正、林杜、芦丽丹和马晓峰合力编写而成，示例极为丰富、相当实用。无论是对大专院校的大学生、研究生，还是对科研院所的研究人员，都无疑是一本难得的三维图形技术指导和参考书。本书亦可作为从事计算机图形工作的一般人员的三维图形技术引导，此外，也非常适合作为 OpenGL 三维图形编程的培训教程。

为了便于用户使用，本书附有示例程序的光盘。所附程序盘按书中章节分目录存放，每章中的例子程序以 examxx 的名称顺序命名，分别表示书中具有完整代码的程序。程序使用 Visual C++ 5.0 以上的版本编译，运行环境是 Windows 98 或 Windows NT。书中的程序均由作者调试通过。

目 录

第 1 章 OpenGL 编程基础知识 1

1.1 图形学的基本发展状况 1

1.1.1 蓬勃发展的计算机图形学 1

1.1.2 计算机图形学的广泛应用 2

1.2 OpenGL 概述 3

1.2.1 OpenGL 基本概念 4

1.2.2 OpenGL 命令语法及各种状态的含义 7

1.3 运用辅助库创建规则几何对象 8

1.3.1 辅助库简介 8

1.3.2 辅助库分类 8

1.3.3 辅助库应用示例 8

1.4 OpenGL 辅助库函数详解 16

1.4.1 窗口初始化函数 16

1.4.2 窗口处理和事件处理函数 17

1.4.3 定义场景绘制循环函数 17

1.4.4 三维物体绘制函数 18

1.4.5 颜色索引表装入函数 19

1.4.6 空闲事件处理函数 19

1.5 Windows 环境下 OpenGL 编程简介 19

1.5.1 OpenGL 在 Windows 下的运行机制 19

1.5.2 Windows 下新增 OpenGL 相关函数 20

1.5.3 VC 环境下基于 OpenGL 的编程步骤 20

1.5.4 建立三维实体模型 22

第 2 章 Win32 环境下 OpenGL 编程概论 25

2.1 Windows 环境下 OpenGL 基本程序结构 25

2.1.1 Windows 环境下 OpenGL 工作机制 25

2.1.2 OpenGL 与 Windows 的绘图方式 25

2.1.3 像素格式的结构和参数设置 26

2.2 Windows 环境下 OpenGL 编程方法 28

2.2.1 示例源程序 28

2.2.2 建立 GDI 窗口 35

2.2.3 建立用于 OpenGL 绘制的窗口 36
2.2.4 用 OpenGL 绘图 37
2.2.5 释放设备描述表和绘制描述表 39
2.3 利用 Visual C++ 6.0 的 AppWizard 编写 OpenGL 应用程序 39
2.3.1 创建一个新工程 40
2.3.2 设置编译环境 43
2.3.3 加入 OpenGL 库 43

第 3 章 Win32 环境下 OpenGL 相关函数及结构 44

3.1 与 OpenGL 相关的 Win32 函数 44
3.1.1 ChoosePixelFormat 函数 44
3.1.2 DescribePixelFormat 函数 46
3.1.3 GetPixelFormat 函数 47
3.1.4 GetEnhMetaFilePixelFormat 函数 48
3.1.5 SetPixelFormat 函数 48
3.1.6 SwapBuffers 函数 49
3.2 与 OpenGL 相关的 wgl 函数 50
3.2.1 wglCreateContext 函数 50
3.2.2 wglCreateLayerContext 函数 50
3.2.3 wglcopyContext 函数 51
3.2.4 wglDeleteContext 函数 52
3.2.5 wglDescribeLayerPlane 函数 52
3.2.6 wglGetcurrentContext 函数 52
3.2.7 wglGetCurrentDC 函数 53
3.2.8 wglGetLayerPaletteEntries 函数 53
3.2.9 wglGetProcAddress 函数 53
3.2.10 wglMakeCurrent 函数 54
3.2.11 wglRealizeLayerPalette 函数 54
3.2.12 wglSetLayerPaletteEntries 函数 55
3.2.13 wglShareLists 函数 55
3.2.14 wglSwapLayerBuffers 函数 56
3.2.15 wglUseFontBitmaps 函数 56
3.2.16 wglUseFontOutlines 函数 58
3.3 与 OpenGL 相关的结构 59
3.3.1 GLYPHMETRICSFLOAT 结构 59
3.3.2 LAYERPLANEDESCRIPTOR 结构 60
3.3.3 PIXELFORMATDESCRIPTOR 结构 62
3.3.4 POINTFLOAT 结构 64

第 4 章 OpenGL 与 Visual C++ 6.0 事件编程 66

4.1 简单编程思路 66
4.1.1 程序功能简介 66
4.1.2 简要实现步骤 66
4.2 实例过程 67
4.2.1 建立一个单文档 67
4.2.2 用资源编辑器制作一个命令操作对话框 67
4.2.3 为命令操作对话框建立一个视图类 67
4.2.4 在 CMainFrame 中创建一个 CSplitterWnd 类 68
4.2.5 编写命令操作对话框的事件处理程序 69
第 5 章 基本几何图元的定义 84
5.1 绘制初始化命令 84
5.2 基本图元的绘制 85
5.2.1 点、线与多边形的简单定义 85
5.2.2 特殊多边形的绘制方法 87
5.2.3 点、线与多边形属性的设置 88
5.2.4 法向量的计算方法 89
5.2.5 利用简单几何图元构造较复杂形体 90
5.3 基本图元绘制实例程序 93
第 6 章 坐标变换 109
6.1 坐标变换的基本概念 109
6.2 通用变换命令 112
6.3 视图和造型变换 113
6.3.1 变换的顺序 113
6.3.2 造型变换 114
6.3.3 视图变换 117
6.4 投影变换 119
6.4.1 透视投影 119
6.4.2 正交投影 121
6.5 视口变换 122
6.5.1 定义视口 122
6.5.2 变换 z 坐标 123
6.6 操作矩阵栈 123
6.7 附加裁剪面 125
6.8 利用几种变换综合绘制机器人手臂 127
第 7 章 OpenGL 颜色 140
7.1 色彩视觉 140
7.2 计算机颜色 141

7.2.1 颜色生成原理 141
7.2.2 RGB 颜色模型 142
7.3 颜色模式 142
7.3.1 RGBA 颜色模式 143
7.3.2 颜色索引模式 143
7.3.3 两种模式的应用场合 143
7.4 指定颜色和明暗处理模型 144
7.4.1 在 RGBA 模式下指定颜色 144
7.4.2 在颜色索引模式下指定颜色 145
7.5 颜色索引模式下指定颜色实例过程 145
7.5.1 在颜色索引模式下指定颜色 145
7.5.2 包含文件及全局变量 146
7.5.3 绘制圆环 147
7.5.4 设置投影方式 149
7.5.5 初始化 OpenGL 环境 149
7.5.6 变换后重绘圆环 150
7.5.7 窗口改变大小 150
7.5.8 设置逻辑调色板 151
7.5.9 设置像素格式 153
7.5.10 主窗口事件处理函数 154
7.5.11 主函数 156
7.5.12 程序执行结果 158
第 8 章 光照高级编程 159
8.1 真实世界与 OpenGL 中的光照 159
8.1.1 光照分量 160
8.1.2 材质颜色 160
8.1.3 光线与材质的 RGB 值 161
8.2 简单示例 161
8.3 创建光源 163
8.3.1 颜色 164
8.3.2 位置与衰减 165
8.3.3 聚光灯 165
8.3.4 多光源 166
8.3.5 控制光源位置和方向 167
8.4 选择光照模型 171
8.4.1 全局环境光 171
8.4.2 局部和无穷远视点 171
8.4.3 双面光照 172

8.4.4 启用光照 172
8.5 定义材质特性 172
8.5.1 漫射和环境反射 173
8.5.2 镜面反射 174
8.5.3 发射光 174
8.5.4 改变材质特性 174
8.6 光照中的数学 179
8.6.1 材质发射光 179
8.6.2 放缩的全局环境光 179
8.6.3 光源的贡献 179
8.6.4 合成 181
8.7 颜色索引模式中的光照 181
8.8 发光小球碰撞实验 182

第 9 章 读取三维图形数据 194

9.1 三维图形数据文件格式简介 194
9.1.1 Wavefront 模型简介 194
9.1.2 3DS 文件格式 196
9.1.3 各种 3D 数据模型相互转换 203
9.2 在 OpenGL 中读取和操作 3DS 模型文件 203

第 10 章 位图、字体和图像编程 236

10.1 位图和字体 236
10.1.1 OpenGL 对字符显示的支持 236
10.1.2 当前光栅位置 238
10.1.3 绘制位图 238
10.1.4 字体和显示列表 239
10.1.5 定义和使用一个完整的字体 240
10.2 OpenGL 中生成图像的打印 247
10.2.1 OpenGL 中图像的绘制 247
10.2.2 OpenGL 中生成图像打印的基本步骤 251
10.2.3 实例过程 251
10.2.4 程序执行结果 260

第 11 章 纹理映射高级编程 261

11.1 定义纹理 261
11.2 控制纹理滤波 262
11.2.1 滤波 262
11.2.2 重复与缩限 263
11.2.3 映射方式 263

11.2.4 纹理坐标定义 263
11.2.5 自动产生纹理坐标 263
11.3 用位图文件创建纹理 270
11.3.1 调用函数来创建纹理 270
11.3.2 从文件中直接读写 272

第 12 章 各种特殊效果的实现 273

12.1 融合 273
12.1.1 源和目的因子 274
12.1.2 颜色融合的应用 275
12.1.3 一个颜色融合的例子 275
12.1.4 利用深度缓存的三维融合 277
12.2 反走样 283
12.2.1 反走样点或线 285
12.2.2 反走样多边形 289
12.3 雾化 292
12.3.1 雾的用法 292
12.3.2 雾方程 295

第 13 章 帧缓存技术与 OpenGL 动画 299

13.1 缓存及其使用 299
13.1.1 OpenGL 中的各种缓存 300
13.1.2 清除缓存 301
13.1.3 选择绘图的颜色缓存 302
13.1.4 屏蔽缓存 303
13.2 检验和操作片断值 303
13.2.1 剪裁检验 304
13.2.2 a 检验 304
13.2.3 模板检验 305
13.2.4 深度检验 320
13.2.5 融合、抖动和逻辑操作 321
13.3 累积缓存 322

第 14 章 巧用显示列表 324

14.1 显示列表的基本概念 324
14.2 显示列表的建立、执行和管理 327
14.2.1 显示列表的建立 327
14.2.2 显示列表中的存储内容 329
14.2.3 执行显示列表 330
14.2.4 显示列表的层次 332

14.2.5 管理显示列表及其索引 333
14.3 改变封装方式的显示列表 338

第 15 章 参数曲面和曲线编程 340

第 16 章 选择与反馈实现图形交互 350

16.1 选择模式 350
16.1.1 选择的基本步骤 351
16.1.2 建立名字栈 352
16.1.3 命中记录 352
16.1.4 选择的例子 353
16.1.5 拾取 353
16.1.6 利用选择编程的提示 354
16.2 反馈模式 355
16.2.1 反馈数组 356
16.2.2 在反馈模式下利用标记 357
16.2.3 反馈的例子 357
16.2.4 命中记录 359
16.3 一个选择的例子 359
16.4 拾取 363
16.4.1 简单示例 363
16.4.2 从多个名称和一个层次结构中进行拾取 367

第 17 章 用 OpenGL 编写屏幕保护程序 373

17.1 Windows 环境下的屏幕保护程序 373
17.1.1 设置屏幕保护程序 373
17.1.2 安装屏幕保护程序 375
17.2 屏幕保护程序编写的基本步骤 376
17.2.1 屏幕保护响应程序 376
17.2.2 屏幕保护设置程序 382
17.2.3 图形绘制程序 382
17.3 屏幕保护程序实例 384
17.3.1 屏幕保护程序主程序 384
17.3.2 图形绘制程序 386

第 18 章 建筑物虚拟漫游编程 394

18.1 虚拟现实系统简介 394
18.1.1 虚拟现实技术的产生、基本特征及构成 394
18.1.2 多维信息空间 395
18.1.3 实现虚拟现实的软硬件平台 396

18.1.4 虚拟环境人机交互系统 397
18.2 虚拟漫游技术简介 398
18.2.1 虚拟漫游系统示例 398
18.2.2 建筑物虚拟漫游 399
18.3 虚拟建筑物漫游编程实例 400
18.3.1 辐射度方法简介 400
18.3.2 数据存储结构 400
18.3.3 读取辐射度数据 402
18.3.4 纹理映射相关程序 404
18.3.5 绘制程序 407
18.3.6 漫游动作控制 409
第 19 章 OpenGL 新版本的新增功能 412
19.1 新增纹理处理功能 412
19.1.1 新增纹理格式 412
19.1.2 使用纹理环境的新方法 413
19.1.3 纹理代理机制 415
19.1.4 使用纹理物体 418
19.1.5 使用子纹理 421
19.1.6 从帧缓存中装入纹理 422
19.1.7 其他新的纹理扩展函数 423
19.2 多边形偏移功能 424
19.2.1 使用多边形偏移功能 424
19.2.2 多边形偏移编程示例 425
19.3 OpenGL 新版本顶点数组 428
19.3.1 指定顶点数组 428
19.3.2 变换独立的数组单元 430
19.3.3 连接不同数组的数组单元 430
19.3.4 通过建立顶点网格绘制实体 431
19.3.5 使用隔行数组 431
19.3.6 顶点数组扩展功能 432
第 20 章 OpenGL 扩展功能及程序改进 434
20.1 Win32 环境下 OpenGL 扩展集 434
20.1.1 确定扩展集可用 434
20.1.2 编译顶点数组扩展集 437
20.1.3 物体空间顶点拾取扩展集 438
20.1.4 调色板纹理扩展集 439
20.1.5 BGRA 扩展集 440

20.1.6 索引纹理扩展集 441
20.1.7 索引材质扩展集 442
20.1.8 索引功能扩展集 442
20.1.9 索引数组格式扩展集 443
20.2 OpenGL 程序性能的改进 444
20.2.1 改进程序流程 444
20.2.2 改进几何体子系统 446
20.2.3 改进栅格化子系统 449
20.2.4 提高图像和像素绘制性能 451
20.2.5 改进动画性能 453
20.2.6 进行时间测定 454
20.2.7 SGI OpenGL for Windows 库使用技巧 455

第 1 章　OpenGL 编程基础知识

随着计算机软、硬件突飞猛进的发展，计算机图形学在各个行业的应用也得到迅速普及和深入。目前，计算机图形学已进入三维时代，三维图形在人们周围无所不在。魅力无穷的计算机广告，扣人心弦的三维游戏，声势浩大、震撼人心的影视特效，无一不是三维图形大显身手的得意之作。

科学可视化、计算机动画和虚拟现实已成为近年来计算机图形学的三大热门话题，而这三大热门话题的技术核心均为三维图形。当前，三维图形已在军事、航空、航天、医学、地质勘探、文化娱乐和艺术造型等方面有着十分广泛的应用。

OpenGL 是一种与硬件、窗口系统和操作系统相独立的一系列 API， 它可以被集成到 UNIX，Windows 95，Windows NT 和 X Window 等窗口系统中。OpenGL 由大量功能强大的图形函数组成，开发人员可以利用这些函数对整个三维图形进行光色渲染，从而方便地绘制出客观世界逼真的三维景象。

以下先简要介绍图形学的基本发展状况，然后通过一个简单的示例向读者展示 OpenGL 的诱人魅力，衷心希望本书能抛砖引玉，引导读者在 OpenGL 的广阔天地中深入探索。

1.1　图形学的基本发展状况

1.1.1　蓬勃发展的计算机图形学

计算机图形学是一个最令人振奋且发展迅速的计算机领域。计算机图形学的发展历史可以追溯到 50 年代末至 60 年代初期。当时的计算机主要应用于科学计算，使用尚不普及，但已开始配置了图形显示器。例如，美国麻省理工学院旋风计算机就配置了由计算机驱动的阴极射线管式的显示器，但不具备人机交互能力。50 年代末期，美国麻省理工学院林肯实验室研制的 SAGE 空中防御系统，就已具备了指挥和控制功能。这个系统能将雷达信号转换为显示器上的图形，操作者可以借用光笔指向屏幕上的目标图形来获得所需要的信息，这一功能的出现预示着交互式图形生成技术的诞生。

1963 年，美国麻省理工学院的 I.E.Sutherland 在他的博士论文中提出了 SKETCHPAD 系统。这一系统用的是 TX.2 型计算机及阴极射线管式图形显示器，它可用光笔在图形显示器上实现选择、定位等交互功能。计算机还可以跟踪光笔，从原来所在的点到所指定的点画直线，或在给定圆心和半径后画圆等。这一系统还引入了分层存储符号和图形元素的数据结构，一幅完整的图可以通过分层调用若干子图来产生。这些基本概念和技术至今仍有用。因此，I.E.Sutherland 的 SKETCHPAD 系统被公认为对交互式图形生成技术的发展

奠定了基础。

在 60 年代，计算机主要以大型机的形式出现，大多数用于科学计算及事务管理。因而以大型计算机连接图形终端成为计算机图形生成技术的运行环境。进入 70 年代以后，由于集成电路技术的发展，计算机硬件性能不断提高、体积缩小、价格降低，特别是廉价的图形输入、输出设备及大容量磁盘的出现，以小型计算机及超级小型机为基础的图形生成技术开始进入市场并形成主流。由于这种系统比起大型计算机来，价格相对便宜，维护使用也比较简单。因而，70 年代以来，计算机图形生成技术在计算机辅助设计、事务管理和过程控制等领域得到了比较广泛的应用。

进入 80 年代以后，工作站的出现，极大地促进了计算机图形学的发展。比起小型计算机来，工作站在图形生成上具有显著的优点。首先，工作站是一个用户使用一台计算机交互作用时，响应时间短；其次，工作站联网后可以共享资源，如，大容量磁盘、高精度绘图仪等；再有，它便于逐步投资、逐步发展、使用寿命较长。因而工作站已经取代小型计算机而成为图形生成的主要环境。目前，工作站正朝着提高速度。扩大容量、提高分辨率、具有标准化的友好用户界面以及多媒体等方向发展。

80 年代后期至今，微机的性能也迅速提高，无论是整数运算速度、内存容量还是磁盘容量，都可以和工作站相媲美，而且大多数微机都配备了高分辨率显示器及窗口管理系统，因此微机也已经成为图形生成的重要环境。由于微机价格比工作站更加便宜，因而受到广大用户的欢迎，更易于普及和推广。随着微机性能的进一步提高，可以预计，它将在计算机图形技术的应用中扮演更为重要的角色。

1.1.2　计算机图形学的广泛应用

计算机图形系统的软硬件性能日益提高，而价格却逐步降低，这必然促使计算机图形生成技术的应用日益广泛。当前计算机图形学已在航空、航天、汽车制造、电子、机械、影视广告、地理信息和数据可视化等领域取得了广泛的应用。

计算机图形学主要应用在设计方面，尤其是工程和建筑系统，现在几乎所有的产品都是由计算机来设计。计算机辅助设计(CAD)方法现在已频繁地应用于大楼、汽车、飞机、轮船、宇宙飞船、计算机、纺织品和许许多多其他产品的设计中。对某些设计来说，设计对象以线框图形式显示出来，以展示出其整个外形及该对象的内部特征。显示线框图可让设计者较快地看到对设计的外形交互修改的结果。CAD 的应用还通常使用动画，如果不需要表面绘制，则可以较快地生成平稳的实时动画。在虚拟现实环境中使用动画确定飞行器的内部结构以及运动时内部构件的行为。在设计完成或将要完成的时候，应用真实感光照模型和曲面绘制技术可给出最终产品的外貌。制造过程也和设计对象的计算机描述联系起来以提供产品的自动化程度。例如，电路板布局可转换为构造该布局的各个过程的描述。某些机械零件按照如何使用机床加工曲面的描述来制造。建筑设计师使用交互式图形技术进行平面布局，给出房间、门、窗、楼梯、橱和柜等的位置及其他特征。面对屏幕上显示的大楼平面布局图，电器设计师可以设计如何布线，安装电源插座以及防火报警系统等。

计算机图形学广泛地应用于美术和商业艺术产品中。美术师使用各种计算机方法，包括专用硬件、艺术家画笔程序和其他绘图软件包和动画软件包等来设计物体的外形及描述

物体的运动。优秀的美术师可以使用各种计算机技术生成图像，他们可以混合使用各种建模软件包、绘图软件和 CAD 软件等。许多商业美术应用(动画及影视广告等)使用逼真照片绘制产品的图片。广告中经常使用动画，电视广告是通过一帧一帧地绘制并存储在一个文件中来生成的。运动的模拟通过在后继帧中的物体相对于前一帧作微小的位移来实现。绘制好动画序列的所有帧以后，将这些帧传送到胶片上或存到视频缓存中重播。电影动画需要顺序播放 24 帧每分，如果在视频上重播动画则需要 30 帧每分。

现在计算机图形学方法常用于制作动画、音乐录像带和电视片。图形场景有时单独显示，有时则和演员及实际场景混合显示。许多电视连续剧中经常使用计算机图形方法。音乐录像片以几种不同方法来使用图形学。图形对象可以被混合进实景中，图形学中的变形技术可被用来将一个人或对象变成另一个。

科学家、工程师、药剂师、商业分析员等常要分析大量的信息或研究特定处理的行为。在巨型机上进行数值计算不断生成包含成千上万数据项的数据文件。同样地，卫星摄像机等也以比解释速度快得多的速度积聚大量数据文件。扫描众多数据以确定趋势及相互关系是一个乏味而低效的过程，但是，如果将这些数据转换为可视形式，则趋势和模式就可以立即显现。存在许多不同类型的数据集，而高效的可视化方法则依赖于数据的特征。一组数据可以包含标量、向量、高次张量或这些数据类型的组合。

尽管在计算机图形学和图像处理中所使用的技术有重叠，但两者有不同的基本操作。在计算机图形学中使用计算机来生成图。而图像处理使用有关技术修改或解释现有图片，比如照片和电视扫描片。图像处理的两个主要应用是：改善图片质量和对视觉信息的机械感觉。

应用计算机图形生成技术产生高精度的地理图形或其他自然资源的图形是另一个重要的应用领域。包括地理图、地形图、矿藏分布图、海洋地理图、气象图和植被分布图等。目前，地理信息管理系统已在很多方面得到广泛的应用，它就是建立在地理图形基础上的信息管理系统。

随着微型计算机及桌上印刷设备的发展，计算机图形学及人机交互技术在办公自动化及电子出版系统中得到广泛的应用。昔日需要提交给专门的印刷机构出版的资料，现在可以在办公室内印刷了。

国外资料表明，在 21 世纪，计算机图形生成技术的应用仍将不断发展，计算机图形系统的市场将以每年 15%~20%的递增速度不断扩大。

1.2 OpenGL 概述

为了用计算机模拟人们生活的三维世界，必须在三维空间中描绘这些物体。对于这个充满了信息的世界，人们渴望尽快地理解并运用这些信息，因此需要一种直接的、易于理解的形式来表达这些信息，计算机图形学在这方面起着举足轻重的作用。许多人在这方面作出了卓越的贡献，例如，长期以来，从事三维图形开发的技术人员都编写了大量的诸如矩阵变换、外部设备访问等函数。然而，他们做的很多工作与自己的目标关系都不是很大，在这方面费脑筋着实是极大的脑力浪费。OpenGL 提供了一种直观的编程环境，它提供的

一系列函数大大简化了三维图形程序。

1.2.1 OpenGL 基本概念

在计算机发展的初期，人们就开始从事计算机图形的开发。直到计算机软硬件和计算机图形学高度发达的 80 年代末 90 年代初，人们发现复杂的数据以视觉的形式表现出来是最容易理解的，因而三维图形得以迅猛发展，于是各种三维图形工具软件包相继推出，如 GL，RenderMan 等。这些三维工具软件包有的侧重于使用方便，有的侧重于绘制效果与应用软件的连接，但没有一种三维工具软件包在交互式三维图形建模能力和编程方便程度上能够和 OpenGL 相比。OpenGL 经过对 GL 的进一步发展，更加灵活方便地实现了二维和三维的高级图形技术，在性能上表现得异常优越。它包括建模、变换、光线处理、色彩处理、动画以及更先进的能力，如纹理映射、物体运动模糊效果和雾化效果等。OpenGL 的这些能力为实现逼真的三维绘制效果，建立交互的三维场景提供了良好的条件。

OpenGL 是近几年发展起来的一个性能卓越的三维图形标准，它是在 SGI 等多家世界闻名的计算机公司的倡导下，以 SGI 的 GL 三维图形库为基础制定的一个通用共享的开放式三维图形标准。目前，包括 Microsoft，SGI，IBM，DEC，SUN 和 HP 等大公司都采用了 OpenGL 作为三维图形标准，许多软件厂商也纷纷以 OpenGL 为基础开发出自己的产品，其中比较著名的产品包括动画制作软件 SoftImage 3D 和 3D Studio MAX、仿真软件 Open Inventor、VR 软件 World Tool Kit、CAM 软件 ProEngineer、GIS 软件 ARC/INFO 等。值得一提的是，随着 Microsoft 公司在 Windows NT 和最新的 Windows 95/98 中提供了 OpenGL 标准及 OpenGL 三维图形加速卡的推出，OpenGL 将在微机中有广泛地应用，同时也为广大用户提供了在微机上使用以前只能在高性能图形工作站上运行的各种软件的机会。也就是说它使得在微机上实现三维图形的生成与显示成为可能。同时由于 OpenGL 是开放的图形标准，用户原先在 UNIX 下开发的 OpenGL 图形软件很容易移植到微机上的 Windows 9x/NT 上。

OpenGL 独立于硬件设备、窗口系统和操作系统等，许多计算机公司已经把 OpenGL 集成到各种操作系统和窗口系统中，其中窗口系统有 X 窗口系统，Windows 等，操作系统包括 Unix，Windows NT 和 Windows 9x 等。为了实现一个完整功能的图形处理系统，设计一个与 OpenGL 相关的系统结构为：最底层是图形硬件，第二层为操作系统，第三层为窗口系统，第四层为 OpenGL，第五层为应用软件。

OpenGL 实际上是一个开放的三维图形软件包，它独立于窗口系统和操作系统，以它为基础开发的应用程序可以十分方便地在各种平台间移植。OpenGL 可以与 Visual C++紧密接口，便于实现机械手的有关计算和图形算法，可保证算法的正确性和可靠性。OpenGL 使用简便、效率高。它具有七大功能，介绍如下：

- 建模

OpenGL 图形库除了提供基本的点、线、多边形的绘制函数外，还提供了复杂的三维物体(球、锥、多面体和茶壶等)以及复杂曲线和曲面(如 Bezier、Nurbs 等曲线或曲面)绘制函数。

● 变换

OpenGL 图形库的变换包括基本变换和投影变换。基本变换有平移、旋转、缩放和镜像 4 种变换，投影变换有平行投影(又称正射投影)和透视投影两种变换。其变换方法与机器人运动学中的坐标变换方法完全一致，有利于减少算法的运行时间，提高三维图形的显示速度。

● 颜色模式设置

OpenGL 颜色模式有两种，即 RGBA 模式和颜色索引(Color Index)。

● 光照和材质设置

OpenGL 光有辐射光(Emitted Light)、环境光(Ambient Light)、漫反射光(Diffuse Light)和镜面光(Specular Light)。材质用光反射率表示。场景(Scene)中物体最终反映到人眼的颜色是光的红、绿、蓝分量与材质红、绿、蓝分量的反射率相乘后形成的颜色。

● 纹理映射(Texture Mapping)

利用 OpenGL 纹理映射功能可以十分逼真地表达物体表面细节。

● 位图显示和图像增强

图像功能除了基本的拷贝和像素读写外，还提供融合(Blending)、反走样(Antialiasing)和雾(Fog)的特殊图像效果处理。以上 3 条可使被仿真物更具真实感，增强图形显示的效果。

● 双缓存(Double Buffering)动画

双缓存即前台缓存和后台缓存，简而言之，后台缓存计算场景、生成画面，前台缓存显示后台缓存已画好的画面。

此外，利用 OpenGL 还能实现深度暗示(Depth Cue)、运动模糊(Motion Blur)等特殊效果，从而实现消隐算法。

OpenGL 是一个硬件发生器的软件接口，其主要目的是将二维、三维物体绘制到一个帧缓存里，它包括几百个图形函数，开发者可以利用这些函数建立三维模型和进行三维实时交互。大多数 OpenGL 系统要求图形硬件系统中至少包含一个帧缓存。OpenGL 的图形函数不要求开发者把三维物体模型数据写成固定的数据格式，这样开发者不但可以使用自己的数据，而且可以利用其他不同格式的数据源，如 3DS 格式的文件等(第 9 章将介绍如何在 OpenGL 中获取各种格式的数据源)。这种灵活性大大地节省了开发者的时间，提高了软件的开发效益。使用 OpenGL，从事三维图形开发的技术人员就不需要在自己的程序里编写许多不必要的矩阵变换、外部设备访问函数了。

下面先介绍 OpenGL 的基本原理，解释 OpenGL 自身的一些特有概念。

● 过程性而非描述性

OpenGL 提供对二、三维图形基本操作非常直接的控制，包括对诸如变换矩阵、光照方程系数、反走样方法和像素更新操作符等参数的指定。但是，它不提供对复杂几何对象的描述或建模的手段。因此，发布 OpenGL 命令就是要指定怎样产生一个特定的结果，而不是确切说明结果应该如何。也就是说，OpenGL 是过程性而非描述性的。由于这种过程性，了解 OpenGL 工作方式(例如，它的操作顺序)将有助于完全理解怎样使用 OpenGL 库。

● 执行模式

OpenGL 命令的解释模式是客户/服务器式的。应用程序(客户)发布命令，命令被

OpenGL(服务器)解释和处理。服务器可以运行在与客户相同或不同的计算机上。基于这一点，OpenGL 是网络透明的。服务器可维护许多 OpenGL 正文，每个都是封装的 OpenGL 状态。客户可连到这些正文中的任何一个。这要求网络协议可以通过扩展现有协议或使用一个独立协议来实现。没有提供获取用户输入的 OpenGL 命令。

分配帧缓存资源的窗口系统最终控制 OpenGL 命令在帧缓存上的作用效果。窗口系统决定任何给定时刻 OpenGL 可以访问帧缓存的哪些部分，并通知 OpenGL 这些部分是怎样组织的。因此，配置帧缓存或初始化 OpenGL 的工作不是由 OpenGL 命令完成，而是由窗口系统完成。帧缓存的初始化是结合窗口系统在 OpenGL 外完成的，窗口系统为 OpenGL 绘制分配窗口时实现 OpenGL 初始化。

- 图元与命令

OpenGL 能够绘制的图元包括点、线段和多边形，OpenGL 可以在这几种图元模式之间选择。用户可以独立地控制图元模式，即设定一种模式不影响其他模式。模式的选择、图元的定义以及其他 OpenGL 操作都是通过调用相应函数来实现的。

图元被定义为一组顶点。顶点是构成线和多边形基本元素的。与顶点相关联的数据包括其坐标、颜色、法向量、纹理坐标、边标识等，在通常情况下，OpenGL 以相同的方法独立地、顺序地处理每个顶点及其关联数据。唯一例外的情况是，如果一组顶点必须被裁剪，那么这个图元将被调整到与指定的区域匹配。这种情况下，顶点数据将被修改并产生一些新的顶点。这类裁剪基于顶点组所描述的图元。

OpenGL 命令总是按接收到的顺序进行处理，也就是说先定义的图元画完后才会执行随后的命令，状态查询命令返回在调用它之前所有发给 OpenGL 的命令完全执行后的相应数据。

- 绘制方式

OpenGL 的绘制过程多种多样，内容十分丰富，主要提供了以下对三维物体的绘制方式：

- ◆ 线框绘制方式(wire frame)：这种方式仅绘制三维物体的网格轮廓线。
- ◆ 深度优先线框绘制方式(depth cued)：用线框方式绘图，使远处的物体比近处物体暗一些，以模拟人眼看物体的效果。
- ◆ 反走样线框绘制方式(antialiased)：用线框方式绘图，绘制时采用反走样技术以减少图形线条的参差不齐。
- ◆ 平面明暗处理方式(flat shading)：对模型的平面单元按光照度进行着色，但不进行光滑处理。
- ◆ 光滑明暗处理方式(smooth shading)：对模型按光照绘制的过程进行光滑处理，这种方式更接近于现实。
- ◆ 加阴影和纹理的方式(shadow，texture)：在模型表面贴上纹理甚至加上光照阴影效果，使三维场景像照片一样逼真。
- ◆ 运动模糊绘制方式(motion blured)：模拟物体运动时人眼观察所感觉到的动感模糊现象。
- ◆ 大气环境效果(atmosphere effects)：在三维场景中加入雾等大气环境效果，使人身临其境。
- ◆ 深度域效果(depth of effects)：类似于照相机镜头效果，模型在聚焦点处清晰，

反之则模糊。

1.2.2　OpenGL 命令语法及各种状态的含义

OpenGL 包含 100 多个库函数，这些函数都按一定的格式来命名。在 Windows NT 下，OpenGL 除了具有基本的 OpenGL 函数外，还支持其他 4 类函数：

- OpenGL 实用库：43 个函数，每个函数以 glu 开头。
- OpenGL 辅助库：31 个函数，每个函数以 aux 开头。
- Windows 专用库函数：6 个函数，每个函数以 wgl 开头。
- Win32 API 函数：5 个函数，函数前面没有专门的前缀。

在 OpenGL 中有 115 个核心函数，这些函数是最基本的，它们可以在任何 OpenGL 的工作平台上应用。这些函数用于建立各种各样的形体，产生光照效果，进行反走样以及纹理映射，进行投影变换等。由于这些核心函数有多种形式并能够接受不同类型的参数，所以这些函数可以派生出 300 多个函数。

这些基本函数的命名规则为：以前缀 gl 开头，组成命令名的每个字的开头用大写字母，例如，glClearColor()。

OpenGL 定义的符号常数以 GL_开头，全部用大写字母，各个字之间用下划线分隔。例如，GL_COLOR_BUFFER_BIT。

OpenGL 定义的数据类型以 GL 开头，表 1.1 列出了各种数据类型。

表1.1　OpenGL函数中字符与数据类型

字符	数据类型	相应与 C 的数据类型	OpenGL 中定义
b	8 位整数	signed char	Glbyte
s	16 位整数	short	Glshort
i	32 位整数	int	GLint，　Glsizei
f	32 位浮点数	float	GLfloat，　Glclampf
d	64 位浮点数	double	GLdouble，　Glclampd
ub	8 位无符号整数	unsigned char	GLubyte，　Glboolean
us	16 位无符号整数	unsigned short	Glushort
ui	32 位无符号整数	unsigned int	GLuint，　GLenum，　GLbitfield
		void	Glvoid

例如，void glVertex<n>{sifd}[v](TYPE x，　TYPE y);

TYPE 为 sifd 数据类型中的一个。

n　说明命令参数的个数，这里可以为 2 或 3。

v　指定命令为向量形式。

例如，利用 32 位浮点数据类型命令的非向量和向量形式分别为：

```
void glVertex2f(GLfloat x, GLfloat y);
void glVertex2fv(GLfloat v[2]);
```

1.3 运用辅助库创建规则几何对象

1.3.1 辅助库简介

OpenGL 中提供了许多图形函数，但是没有窗口函数以及读取键盘和鼠标事件的函数。对于 OpenGL 初学者来说，要编写一个完整的 OpenGL 程序就比较困难了，辅助库可以帮助用户度过这一难关。有时(例如，讲解光照基本原理时)用户希望能够利用简单的函数生成一些通用的几何模型(如球、锥体和圆柱体等)，以便在程序中将注意力集中在各种 OpenGL 处理上，然而 OpenGL 只提供了基本几何图元：点、线、多边形等的绘制，不能满足用户的需要，这时就可以用辅助库。

OpenGL 辅助库中提供了一些基本的窗口管理和基本三维图形绘制函数，可以帮助用户方便地创建完整的 OpenGL 应用程序。不过对于复杂的应用程序以及一些深入的需求，则辅助库远远不能满足要求。辅助库的主要目的是帮助初学者尽快地掌握 OpenGL 的精髓。

1.3.2 辅助库分类

辅助库大致分为以下 6 类：

- 窗口初始化函数。
- 窗口处理和事件处理函数。
- 定义场景绘制循环函数。
- 三维物体绘制函数。
- 颜色索引表装入函数。
- 空闲事件处理函数。

为了使用户比较全面地了解辅助库的内容，下面设计了一个程序，该程序尽可能涵盖辅助库中的所有函数。该程序实现的功能是：定义键盘的功能，0~9 及 a 分别表示选择 11 种基本形体的一种进行绘制；定义鼠标的功能为，左键按下表示选择实心显示模式，右键按下表示选择网状显示模式；空闲事件的功能定义为，反复地对形体进行放大或缩小。程序中用到了上面列出的除颜色索引装入函数以外的其他所有函数。

1.3.3 辅助库应用示例

- 包含文件及函数原型

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glu.h>
```

```
#include <GL/glaux.h>
void MyInit(void);
void DrawScene(int, int);
void CALLBACK Display(void);
void CALLBACK MyReshape(GLsizei w, GLsizei h);
void CALLBACK SetSolid(AUX_EVENTREC *);
void CALLBACK SetWire(AUX_EVENTREC *);
void CALLBACK SetScale(void);
void CALLBACK Key_0(void);
void CALLBACK Key_1(void);
void CALLBACK Key_2(void);
void CALLBACK Key_3(void);
void CALLBACK Key_4(void);
void CALLBACK Key_5(void);
void CALLBACK Key_6(void);
void CALLBACK Key_7(void);
void CALLBACK Key_8(void);
void CALLBACK Key_9(void);
void CALLBACK Key_a(void);
```

- 全局变量

```
GLint Width = 0;                // 窗口宽度
GLint Height = 0;               // 窗口高度
GLint ShapeType, Solid = 1;     // 形体类型及是否实心
GLfloat Scale = 1.0f;           // 形体缩放比例
```

- 初始化函数：定义材质与光照

```
void MyInit (void)
{
   GLfloat diffuse[] = { 0.3f, 0.6f, 0.9f, 1.0f };
   GLfloat specular[] = { 0.8f, 0.8f, 0.8f, 1.0f };

   GLfloat position_one[] = { 1.0f, 1.0f, 1.0f, 0.0f };
  GLfloat position_two[] = {-1.0f, 1.0f, 1.0f, 0.0f };

   // 定义材质
   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
   glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);
   glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 64.0f);

   // 定义 light0 和 light1
```

```
        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
        glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
        glLightfv(GL_LIGHT0, GL_POSITION, position_one);
        glLightfv(GL_LIGHT1, GL_POSITION, position_two);

        // 使用双面光照
        glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
        // 激活光照与深度检测
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHT1);
        glEnable(GL_LIGHTING);
        glEnable(GL_DEPTH_TEST);
    }
```

- 显示响应函数：调用几何形体绘制

程序会循环地调用该函数进行绘制，因此当其他程序中修改了以下绘制程序中的一些参数时，无须调用该函数，程序会自动地用该函数重新进行绘制。

以下程序中先指定视口、清除缓存，经过一系列坐标变换后，再绘制几何形体，然后交换前后缓存，将绘制的内容显示到屏幕上。

```
void CALLBACK Display(void)
{
   glViewport( 0, 0, Width, Height);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glRotatef( 30.0f, 1.0f, 0.0f, 0.0f );
   glRotatef( 30.0f, 0.0f, 1.0f, 0.0f );
   glScalef( Scale, Scale, Scale );
   DrawScene( ShapeType, Solid );
   glFlush();
   glPopMatrix();
   auxSwapBuffers();
}
```

- 当窗口大小发生变化时，相应地调整坐标系

```
void CALLBACK MyReshape(GLsizei w, GLsizei h)
{
   Width = w;
   Height = h;

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
        if( w <= h ) //  宽度小于高度时
        glOrtho(-2.0f,  2.0f, -2.0f*(GLfloat)h/(GLfloat)w,
             2.0f*(GLfloat)h/(GLfloat)w, -2.0f,  2.0f);
      else  //  宽度大于高度时
        glOrtho(-2.0f*(GLfloat)w/(GLfloat)h,
             2.0f*(GLfloat)w/(GLfloat)h, -2.0f, 2.0f, -2.0f, 2.0f);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
    }
```

- 场景绘制函数

该函数绘制辅助库中的所有基本几何体。其中 ShapeType 参数决定绘制何种形体，Solid 参数决定是否绘制成实心体。这里调用了目前辅助库中的提供所有形体绘制函数。细心的用户也许会发现，程序运行的结果中可能碰到了辅助库中的一个小错误，即当绘制圆柱和圆锥时，从实心到网状或从网状到实心的切换不起作用。也就是说若首先绘制的是一个实心的圆柱或圆锥，则无法再绘制一个网状的圆柱或圆锥，同一帧画面中也无法同时显示实心和网状的圆柱或圆锥。辅助库并不太常用，因此也不必担心，不过还是希望 OpenGL 的下一个版本中能改掉这个小错误。

DrawScene 函数是被 Display 函数调用来产生实际形体绘制的，它与其他函数一样无需是回调的(CALLBACK)。

```
void DrawScene(int ShapeType,  int Solid)
{
  switch(ShapeType)
  {
    case 0:
      if ( Solid ) auxSolidSphere(1.0);
         else auxWireSphere(1.0);
      break;
    case 1:
         if ( Solid ) auxSolidCube(1.0);
         else auxWireCube(1.0);
      break;
    case 2:
         if ( Solid ) auxSolidBox(1.5,  1.0,  1.0);
         else auxWireBox(1.5,  1.0,  1.0);
      break;
    case 3:
         if ( Solid ) auxSolidTorus(1.0,  1.0);
         else auxWireTorus(1.0,  1.0);
      break;
```

```
        case 4:
            if ( Solid ) auxSolidCylinder(1.0, 2.0);
            else auxWireCylinder(1.0, 2.0);
          break;
        case 5:
            if ( Solid ) auxSolidIcosahedron(1.0);
            else auxWireIcosahedron(1.0);
          break;
        case 6:
            if ( Solid ) auxSolidOctahedron(1.0);
            else auxWireOctahedron(1.0);
          break;
        case 7:
            if ( Solid ) auxSolidTetrahedron(1.0);
            else auxWireTetrahedron(1.0);
          break;
        case 8:
            if ( Solid ) auxSolidDodecahedron(1.0);
            else auxWireDodecahedron(1.0);
          break;
        case 9:
            if ( Solid ) auxSolidCone(1.0, 1.0);
            else auxWireCone(1.0, 1.0);
          break;
        case 10:
            if ( Solid ) auxSolidTeapot(1.0);
            else auxWireTeapot(1.0);
          break;
      }
    }
```

- 键盘事件处理函数

每一个按键选择一种形体类型，由于程序中定义了控制场景绘制的主循环函数，因此在选择一种形体类型之后，无需调用场景绘制函数，程序仍可根据选择的形体进行绘制。键盘事件处理函数也必须是回调函数，参数应为 void。

```
    void CALLBACK Key_0(void)
    {
        ShapeType = 0;
    }
    void CALLBACK Key_1(void)
```

```
{
   ShapeType = 1;
}

void CALLBACK Key_2(void)
{
   ShapeType = 2;
}

void CALLBACK Key_3(void)
{
   ShapeType = 3;
}

void CALLBACK Key_4(void)
{
   ShapeType = 4;
}

void CALLBACK Key_5(void)
{
   ShapeType = 5;
}

void CALLBACK Key_6(void)
{
   ShapeType = 6;
}

void CALLBACK Key_7(void)
{
   ShapeType = 7;
}

void CALLBACK Key_8(void)
{
   ShapeType = 8;
}
```

```
void CALLBACK Key_9(void)
{
   ShapeType = 9;
}

void CALLBACK Key_a(void)
{
   ShapeType = 10;
}
```

● 鼠标事件处理函数

鼠标事件处理函数定义得非常简单，单击左键表示设置为实心模式；单击右键表示设置为网状模式。与键盘事件处理函数一样，这里只需要修改显示模式，而不必直接调用场景绘制函数进行绘制。鼠标事件处理回调函数的参数为 AUX_EVENTREC *event，用户可以不必去关心。

```
void CALLBACK SetSolid(AUX_EVENTREC *event)
{
   Solid = 1;
}
void CALLBACK SetWire(AUX_EVENTREC *event)
{
   Solid = 0;
}
```

● 空闲事件处理函数

空闲事件处理函数的功能为：先对所绘制的形体进行缩小，当形体缩小到一定程度(此处为 0.1 倍原大小)后，再对形体进行放大，当放大到一定程度(此处为原大小)后，又开始进行缩小，如此反复。程序中使用了一个静态变量来表示放大或缩小状态。

```
void CALLBACK SetScale(void)
{
   static ZoomIn = 0;
   if ( ZoomIn && Scale < 1.0f ){
      Scale = Scale * 1.1f;
     if ( Scale >= 1.0f )
        ZoomIn = 0;
   }
   else {
     Scale = Scale / 1.1f;
     if ( Scale <= 0.1f )
        ZoomIn = 1;
   }
```

```
    Display();
}
```

- 主函数

主函数中作一些初始化显示模式、设置窗口标题、大小、位置及一些事件处理响应函数的工作。控制绘制的主循环函数，鼠标、键盘的事件处理函数以及空闲事件的处理函数也都在这里指定。

```
int main(int argc, char** argv)
{
    auxInitDisplayMode(AUX_DOUBLE | AUX_RGB |
                       AUX_DEPTH16 | AUX_STENCIL);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("AuxLib Demo");
    MyInit();
    auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN, SetSolid);
    auxMouseFunc(AUX_RIGHTBUTTON, AUX_MOUSEDOWN, SetWire);
    auxIdleFunc(SetScale);
    auxKeyFunc(AUX_0, Key_0);
    auxKeyFunc(AUX_1, Key_1);
    auxKeyFunc(AUX_2, Key_2);
    auxKeyFunc(AUX_3, Key_3);
    auxKeyFunc(AUX_4, Key_4);
    auxKeyFunc(AUX_5, Key_5);
    auxKeyFunc(AUX_6, Key_6);
    auxKeyFunc(AUX_7, Key_7);
    auxKeyFunc(AUX_8, Key_8);
    auxKeyFunc(AUX_9, Key_9);
    auxKeyFunc(AUX_a, Key_a);
    auxReshapeFunc (MyReshape);
    auxMainLoop(Display);
    return(0);
}
```

- 程序部分结果(绘制网状球体，如图 1.1 所示)

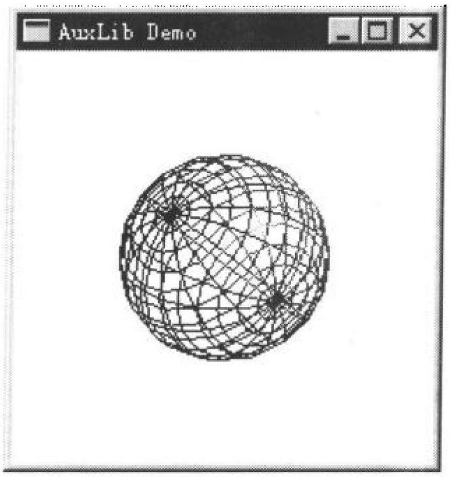

图1.1 辅助库实例部分执行结果

1.4 OpenGL 辅助库函数详解

辅助库作为用户学习 OpenGL 的许多基本知识时非常有用，它可以省去用户许多繁杂的操作，简便地创建基于 OpenGL 的应用。例如，它可以对窗口进行初始化使之可用于 OpenGL 绘制，生成一些常用的基本图形，以及进行简单的交互等。下面对辅助库中的每一类函数作详细的讲解。

1.4.1 窗口初始化函数

此类函数共有 3 个：

- void auxInitWindow(GLbyte *titleString);

表示打开一个窗口，函数的参数是窗口的标题，窗口背景的默认颜色在 RGBA 模式下是黑色，在颜色索引模式下是 0 号调色板的颜色。在打开窗口之前需要先调用 auxInitDisplayMode()和 auxInitPosition()函数。程序运行时在此窗口内按下 Esc 键可以关闭该程序及窗口。

- void auxInitDisplayMode(GLbitfield mask);

用来设置窗口显示模式。窗口的基本模式有 RGBA 或颜色索引模式，单缓存或双缓存模式。也可以用来指定一些附加模式，例如深度缓存、模板缓存或累积缓存。参数 mask 是一组位标志的“或”，有效的参数取值有：AUX_RGBA，AUX_INDEX，AUX_SINGLE，AUX_DOUBLE，AUX_DEPTH，AUX_STENCIL 或 AUX_ACCUM。分别用于指定颜色方式、设置单双缓存，以及激活哪种缓存。例如，设置 RGBA 方式的双缓存窗口以及具有深度和累计缓存的命令为：

```
auxInitDisplayMode(AUX_RGBA|AUX_DOUBLE|AUX_DEPTH|AUX_ACCUM);
```

若不调用该函数，默认情况为：AUX_INDEX|AUX_SINGLE，即颜色索引方式的单缓存窗口。

- void auxInitPosition(GLint x， GLint y， GLsizei width， GLsizei height);

该函数用于设置窗口的大小及位置。参数 x， y 为窗口的左上角屏幕坐标；参数 width 和 height 分别决定窗口的初始宽度和高度，单位均为像素。不调用该函数时，屏幕默认左上角位置在(0，0)处，宽度、高度均为 100。

1.4.2 窗口处理和事件处理函数

窗口处理和事件处理函数由以下 3 个回调函数组成：

- void auxReshapeFunc(void (*function)(GLsizei));

该函数用于指定在窗口形状改变时对绘制产生的影响。参数 function 是一个函数指针，它带有两个参数，即窗口改变后的新宽度和新高度。通常 function 参数中显示裁剪后的新尺寸，重定义投影矩阵，以保证投影后的图像的比例与视点匹配，避免比例失调。程序中若不调用 auxReshapeFunc，默认情况下会调用一个二维正投影矩阵。

- void auxKeyFunction(GLint key，void(*function)(void));

定义键盘事件响应函数。参数 key 对应于键盘上的某个键，辅助库中定义了一些常量，例如，AUX_0~AUX_9，AUX_A~AUX_Z，AUX_a~AUX_z，以及 AUX_LEFT，AUX_RIGHT，AUX_UP，AUX_DOWN 和 AUX_ESCAPE，AUX_SPACE，AUX_RETURN 等。function 函数中定义了当 key 键按下时所执行的操作。

- void auxMouseFunc (GLint button，GLint mode，void(*function) (AUX_EVENTREC *));

定义鼠标事件响应函数。button 的取值可以为：AUX_LEFTBUTTON，AUX_MIDDLE_BUTON 和 AUX_RIGHT_BUTTON，分别表示鼠标左、中和右键。mode 的取值有：AUX_MOUSEDOWN 和 AUX_MOUSEUP 两种，分别表示鼠标按下和释放。Function 参数中定义了当鼠标以 mode 方式作用于 button 上时所执行的操作。它带一个 AUX_EVENTREC 类型指针参数，其中包含鼠标的位置信息等。当 auxMouseFunc 函数被调用时，将为这个结构分配相应的内存。对这个结构的用法如下：

```
void function( AUX_EVENTREC *event)
{
    GLint x, y;
    //  得到鼠标位置
    x = event->data[AUX_MOUSEX];
    y = event->data[AUX_MOUSEY];
    ...
}
```

以上 3 个回调函数应在窗口创建后，进入主循环函数之前调用，才会起作用。

1.4.3 定义场景绘制循环函数

定义场景绘制循环函数是 auxMainloop(void(*displayFunc)(void))。displayFunc 函数指

针指向场景绘制函数。当窗口需要更新时，程序会自动调用它所指向的函数，重新产生绘制。

1.4.4 三维物体绘制函数

属于这一类的函数较多，辅助库总共可以绘制 11 种基本几何图形，每一种图形又包括网状体和实心体两种形式，因此共有 22 个函数，如下所示。

绘制球的函数：

```
void auxWireSphere(GLdouble);
void auxSolidSphere(GLdouble);
```

绘制立方体的函数：

```
void auxWireCube(GLdouble);
void auxSolidCube(GLdouble);
```

绘制长方体的函数：

```
void auxWireBox(GLdouble, GLdouble, GLdouble);
void auxSolidBox(GLdouble, GLdouble, GLdouble);
```

绘制圆纹面的函数：

```
void auxWireTorus(GLdouble, GLdouble);
void auxSolidTorus(GLdouble, GLdouble);
```

绘制圆柱的函数：

```
void auxWireCylinder(GLdouble, GLdouble);
void auxSolidCylinder(GLdouble, GLdouble);
```

绘制二十面体的函数：

```
void auxWireIcosahedron(GLdouble);
void auxSolidIcosahedron(GLdouble);
```

绘制八面体的函数：

```
void auxWireOctahedron(GLdouble);
void auxSolidOctahedron(GLdouble);
```

绘制四面体的函数：

```
void auxWireTetrahedron(GLdouble);
void auxSolidTetrahedron(GLdouble);
```

绘制十二面体的函数：

```
void auxWireDodecahedron(GLdouble);
void auxSolidDodecahedron(GLdouble);
```

绘制圆锥的函数：

```
void auxWireCone(GLdouble, GLdouble);
void auxSolidCone(GLdouble, GLdouble);
```

绘制茶壶的函数：

```
void auxWireTeapot(GLdouble);
```

```
void auxSolidTeapot(GLdouble);
```

1.4.5 颜色索引表装入函数

颜色索引表装入函数为：auxSetOneColor(GLint index， GLfloat red， GLfloat green，GLfloat blue);

因为 OpenGL 没有本身的窗口系统，所以依赖于窗口系统的颜色映射就无法装入查色表。在采用颜色索引模式时，就要用到辅助库提供的这个用 RGB 值定义单个颜色索引的函数。index 表示索引号，red，green 和 blue 用以指定该颜色索引所对应的红、绿、蓝浮点值。

1.4.6 空闲事件处理函数

空闲事件处理函数为：auxIdleFunc(void *func(void))。

定义空闲时间内执行程序的函数，func 指向所要执行的函数功能。当 func 为 NULL 时，func 执行无效。

1.5 Windows 环境下 OpenGL 编程简介

基于 OpenGL 标准开发的应用程序必须运行于 32 位 Windows 平台下，如 Windows NT 或 Windows 95/98 环境，而且运行时还需有动态链接库 OpenGL32.DLL，Glu32.DLL。这两个文件在安装 Windows NT 或 Windows 98 时，已自动装载到 Windows 的 SYSTEM32 目录下；而对于使用 Windows 95 平台的用户，则需手工将两个动态库复制到 Windows 95 目录的 SYSTEM 子目录中。安装了 Windows NT / 9x 和 VC++后，用户就具备了基于 OpenGL 开发三维图形软件的基本条件。

1.5.1 OpenGL 在 Windows 下的运行机制

OpenGL 工作在客户 / 服务器模式下，当客户方(即基于 OpenGL 标准开发的应用程序)向服务器(OpenGL 核心机制)发出命令时，由服务器负责解释这些命令。通常情况下，客户方和服务器是运行在同一台微机上的。由于 OpenGL 的运行机制是客户 / 服务器模式，这使用户能够十分方便地在网络环境下使用 OpenGL，OpenGL 在 WindowsNT 上的这种实现方式通常称为网络透明性。

OpenGL 的图形库函数封装在动态链接库 OpenGL32.DLL 中，客户中的所有 OpenGL 函数调用，都被传送到服务器上，由 WinSrv.DLL 实现功能，再将经过处理的指令发送到 Win32 设备驱动接口(DDI)，从而在计算机屏幕上产生图像。

若使用 OpenGL 图形加速卡，则上述机制中将添加 2 个驱动器：OpenGL 可装载客户模块 OpenGLICD 将安装在客户端；硬件指定模块 DDI 将安装在服务器端，与 WinDDI 同

一级别。

1.5.2 Windows 下新增 OpenGL 相关函数

开发基于 OpenGL 的应用程序，必须先了解 OpenGL 的库函数。OpenGL 函数命令方式十分有规律，每个库函数均有前缀 gl，glu，aux，分别表示该函数属于 OpenGL 基本库、实用库或辅助库。Windows 下的 OpenGL 包含了 100 多个核心函数，均以 gl 作为前缀，同时还支持另外 4 类函数，如下所示。

- OpenGL 实用库函数：43 个，以 glu 作为前缀。
- OpenGL 辅助库函数：31 个，以 aux 作为前缀。
- Windows 专用库函数(WGL)：6 个，以 wgl 作为前缀。
- Win32API 函数(WGL)：5 个，无特定前缀。

OpenGL 的 115 个核心函数提供了最基本的功能，可以实现三维建模、建立光照模型、反走样和纹理映射等。OpenGL 实用库函数在核心函数的上一层，这类函数提供了简单的调用方法，其实质是调用核心函数，目的是减轻开发者的编程工作量；OpenGL 辅助库函数是一些特殊的函数，可以供初学者熟悉 OpenGL 的编程机制，然而使用辅助库函数的应用程序只能在 Win32 环境中使用，可移植性较差，所以应尽量避免使用辅助库函数；Windows 专用库函数(WGL)主要针对 Windows NT / 95 环境的 OpenGL 函数调用；Win32API 函数用于处理像素存储格式、双缓存等函数调用。

1.5.3 VC 环境下基于 OpenGL 的编程步骤

下面介绍在 VC 环境中建立基于 OpenGL 标准的应用程序框架的具体方法与步骤：

(1) 创建项目文件：选择 File|New 菜单项，建立一个名为 MyTest 的项目文件，MyTest 基于单文档(SDI)，View 类基于 CView。

(2) 选择 Build|Setting 菜单项，在 Link 选项组的 Lib 列表框中添加 opengl32.lib，glu32.lib，若需使用 OpenGL 的辅助库函数，则还需添加 glaux.lib。

(3) 选择 View|ClassWizard 菜单项，打开 MFC 对话框，在 ClassName 列表框中选择 CMyTestView 类，进行以下操作：

选择 WM_CREATE 消息，单击 EditCode，将 OpenGL 初始化代码添加到 OnCreate() 函数中：

```
// 定义像素存储格式
PIXELFORMATDESCRIPTOR pfd=
{
  sizeof(PIXELFORMATDESCRIPTOR),
  1,
  PFD_DRAW_TO_WINDOW|PFD_SUPPORT_OPENGL,
  PFD_TYPE_RGBA,
  24,
```

```
        0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0
        32,
        0, 0,
        PFD_MAIN_PLANE,
        0,
        0, 0, 0,
    }
    CCLient dc(this);
    int pixelFormat=ChoosePixelFormat(dc.m_hDC, &pfd);
    BOOL success=SetPixelFormat(dc.m_hDC, pixelFormat, &pfd);
    m_hRC=wglCreateContext(dc.m_hDC);
```

选择 WM_DESTORY 消息，在 OnDestory()中添加以下代码：

```
wglDeleteContext(m_hRC);
```

在 MyTestView.cpp 中，将以下代码添加到 PreCreateWindows()函数中：

```
cs.style|=WS_CLIPCHILDREN|WS_CLIPSIBLINGS;
```

OpenGL 只对 WS_CLIPCHILDREN|WS_CLIPSIBLINGS 类型窗口有效；

在 MyTestView.cpp 中，将以下代码添加到 OnDraw()函数中：

```
wglMakeCurrent(pDC—m_hDC, m_hRC);
DrawScene();    // 用户自定义函数，用于绘制三维场景；
wglMakeCurrent(pDC—m_hDC, NULL);
```

在 MyTestView.cpp 中，添加成员函数 DrawScene()：

```
void CMyTestView::DrawScene()
{
   // 绘制三维场景
}
```

(4) 在 MyTestView.h 中包含以下头文件并添加类成员说明：

```
#include <gl.h>
#include <glu.h>
#include <glaux.h>
```

在 CTestView 类中的 protected：段中添加成员变量声明：

```
HGLRC m_hRC;
```

同时添加成员函数声明：

```
DrawScene();
```

这样，一个基于 OpenGL 标准的程序框架已经构造好，用户只需在 DrawScene()函数中添加程序代码即可。

1.5.4 建立三维实体模型

三维实体建模是整个图形学的基础，要生成高逼真度的图像，首先要生成高质量的三维实体模型。

OpenGL 中提供了十几个生成三维实体模型的辅助库函数，这些函数均以 aux 作为函数名的前缀。简单的模型，如球体、立方体和圆柱等可以使用这些辅助函数来实现，如，auxWireSphere(GLdoubleradius)(绘制一半径为 radius 的网状球体)。但是这些函数难以满足建立复杂三维实体的需要，所以用户可以通过其他建模工具(如 3DS 等)来辅助建立三维实体模型数据库。在三维实体的建模过程中常常可以采用诸如 3DS，Wavefront，WorldUP 等建模工具进行模型的编辑，最后通过将模型数据以某种通用的图形格式(如 3DS，Obj 等文件格式)输出存储，供应用程序使用。

1 定义光照模型和材质

- 光源

OpenGL 提供了一系列建立光照模型的库函数，使用户可以十分方便地在三维场景中建立所需的光照模型。OpenGL 中的光照模型由环境光(AmbientLight)、漫射光(DiffuseLight)和镜面反射光(SpecularLight)等组成，同时还可设置光线衰减因子模拟真实的光源效果。

例如，定义一个黄色光源如下：

```
GLfloat Light_position[]={1.0, 1.0, 1.0, 0.0, };
GLfloat Light_diffuse[]={1.0, 1.0, 0.0, 1.0, };
glLightfv(GL_LIGHT0, GL_POSTTION, light_position); // 定义光源位置
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse); //定义光源漫射光
```

光源必须经过启动后才会影响三维场景中的实体，可以通过以下指令使光源有效：

```
glEnable(LIGHTING); // 启动光照模型;
glEnable(GL_LIGHT0); // 使光源 GL_LIGHT0 有效;
```

OpenGL 中一共可以定义 GL_LIGHT0～GL_LIGHT7 八个光源。

- 材质

OpenGL 中的材质是指构成三维实体的材料在光照模型中对于红、绿、蓝三原色的反射率。与光源的定义类似，材质的定义分为环境、漫射、镜面反射成分，另外还有镜面高光指数、辐射成分等。通过对三维实体的材质定义可以大大提高应用程序所绘制的三维场景的逼真程度。例如：

```
// 设置材质的反射成分
GLfloat mat_ambient[]={0.8, 0.8, 0.8, 1.0};
GLfloat mat_diffuse[]={0.8, 0.0, 0.8, 1.0}; // 紫色
GLfloat mat_specular[]={1.0, 0.0, 1.0, 1.0}; // 镜面高光亮紫色
GLfloat mat_shiness[]={100.0}; // 高光指数
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient); // 定义环境光反射
率
```

```
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);  // 定义漫射光反射率
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);  // 定义镜面光反射率
glMaterialfv(GL_FRONT, GL_SHINESS, mat_shiness);  // 定义高光指数
```

- 材质 RGB 值与光源 RGB 值的关系

OpenGL 中材质的颜色与光照模型中光源的颜色含义略有不同。对于光源，R，G，B 值表示三原色在光源中所占有的比率；而对于材质，R，G，B 的值表示具有这种材质属性的物体对于三原色的反射比率，场景中物体所呈现的颜色与光照模型、材质定义都相关。例如，若定义的光源颜色是(Lr，　Lg，　Lb) = (1.0，　1.0，　1.0)(白光)，物体的材质颜色定义为(Mr，　Mg，　Mb) = (0.0，　0.0，　0.8)，则最终到达人眼的物体颜色应当是(Lr*Mr，Lg*Mg，　Lb*Mb) = (0.0，　0.0，　0.8)(蓝色)。

2. 读取三维模型数据

为了绘制三维实体，首先必须将预先生成的三维实体模型从三维实体模型库中读出。

3. 三维实体绘制

由于 3DS 等文件中对于三维实体的描述是采用三角形面片逼近的方法，而在 OpenGL 函数库中，提供了绘制三角形面片的方法，所以为三维实体的绘制提供了方便。以下提供了绘制三角形面片的方法：

```
glBegin(TRANGLES);                                   // 定义三角形绘制开始
glVertexf((GLfloat)x1,  (GLfloat)y1,  (GLfloat)z1);  // 第一个顶点
glVertexf((GLfloat)x2,  (GLfloat)y2,  (GLfloat)z2);  //第二个顶点
glVertexf((GLfloat)x3,  (GLfloat)y3,  (GLfloat)z3);  // 第三个顶点
glEnd();                                             // 绘制结束
```

为了提高三维实时动画的显示速度，下例利用了 OpenGL 库中的显示列表功能，将三维场景中的实体分别定义为单独的显示列表，预先生成三维实体。在图形显示时，只需调用所需的显示列表即可显示相应的三维实体，而不需要重新计算实体在场景中的坐标，避免了大量的浮点运算。在调用显示列表前所作的旋转、平移、光照和材质的设定都将影响显示列表中的三维实体的显示效果。具体实现方法如下：

```
for(ObjectNo=0;ObjectNo<TotalObj;ObjectNo++)
{
  glNewList(ObjectNo, GL_COMPILE); // 创建第 ObjectNo 个实体的显示列表
  for(FaceNo=0;FaceNo<ObjectNo,  FaceNo++)
  {
   glBegin(GL_TRIANGLES);         // 开始画第 FaceNo 个三角形
   // 定义第 FaceNo 个三角形的第一个顶点的法向量
  glNormal3fv((Glfloat)VertexNormal[ObjectNo][(Face[ObjectNo][Fa
```

```
ceNo][0])]);
      // 定义第 FaceNo 个三角形的第一个顶点的坐标
      glVertex3fv((Glfloat*)Vertex[ObjectNo][(Face[ObjectNo][FaceN
o][0])]);
      // 定义第 FaceNo 个三角形的第二个顶点的法向量;
      // 定义第 FaceNo 个三角形的第二个顶点的坐标;
      // 定义第 FaceNo 个三角形的第三个顶点的法向量;
      // 定义第 FaceNo 个三角形的第三个顶点的坐标;
      glEnd();
  }
  glEndList();
}
…
// 显示三维实体
{
   几何变换、投影变换、裁剪、视区变换、建立光照模型、定义实体材质等操作
}
CallList(ObjectNo);            // 显示第 ObjectNo 个三维实体
```

在这里，要提到的是在绘制三角形面片时，每个三角形顶点的定义前均有该定点法向量的定义，这是为了使得绘制出的三角形面片之间更加光滑、真实。

程序运行结束，需将已建立的显示列表删除，释放所占用的内存空间，可以通过以下函数调用实现删除显示列表：

```
// 删除显示列表
DeleteList(ObjectNo);    // 删除第 ObjectNo 个实体的显示列表
```

第 2 章 Win32 环境下 OpenGL 编程概论

随着微机运算能力和图形绘制能力的逐步提高，目前许多高档微机已经具备了替代以往图形工作站进行复杂图形绘制的能力。由于在微机操作系统的市场上 Microsoft 公司的 Windows 独领风骚，占了绝大部分的微机市场，因此在 Windows 环境下进行软件开发几乎已成为微机上的标准，高级图形软件的开发似乎也不例外。因此在许多场合下要考虑 OpenGL 程序与 Windows 的结合应用，本章就着重介绍 Windows 环境下应用 OpenGL 的方法。

本章内容将分以下两节进行介绍：

- Windows 环境下 OpenGL 基本程序结构

介绍 Windows 环境下进行 OpenGL 编程的基础知识，重点为 OpenGL 与 Windows 相结合的原理和方法。

- Windows 环境下 OpenGL 一般编程方法

通过一个简单的示例，介绍 Windows 下 OpenGL 具体实现过程。

2.1 Windows 环境下 OpenGL 基本程序结构

Windows 环境下进行 OpenGL 三维图形编程与一般的 Windows 图形编程是不同的，OpenGL 在 Windows 环境下的工作方式需通过一种特定的方式实现，本节将分三部分进行介绍。

2.1.1 Windows 环境下 OpenGL 工作机制

Windows 环境下 OpenGL 的绘图功能是通过显示驱动程序(display driver)与图形显示硬件交流的，由显示驱动程序直接控制图形显示硬件。这就是目前市场上流行的图形显示卡都提供一个图形显示驱动程序的原因。对 OpenGL 而言，图形显示卡厂家提供专用的 OpenGL 驱动程序，只有在 OpenGL 驱动程序的支持下 OpenGL 图形命令才得到硬件加速。实现 Windows 下 OpenGL 的应用程序则是通过调用 OpenGL 函数、Windows 的 GDI 函数，以及其他一系列函数来实现的。关于 Windows 下对 OpenGL 扩展的一些函数和功能，将在第 3 章中进行详细讲解。

2.1.2 OpenGL 与 Windows 的绘图方式

OpenGL 的绘图方式与 Windows 的一般的绘图方式是不同的，其区别主要表现在以下

3 个方面：

- Windows 用 GDI 绘图。
- OpenGL 用绘制描述表(Rendering Contexts)绘图。
- OpenGL 使用特殊的像素格式。

本节具体介绍如下这些内容。

1. GDI 与设备描述表

GDI 是 Windows 中二维图形的绘图接口，GDI 绘图内容包括点、线、面以及一系列绘图笔刷等。这些绘图内容可以输出至屏幕内存或者打印机等设备。这种方式都是在 Windows 的设备描述表(即 Device Contexts，简写为 DC)中进行的。在所有的 Windows 图形程序中都需要申请一个 DC 并在 DC 中完成各种绘图操作。

2. 绘制描述表

绘制描述表与设备描述表是连接在一起起作用的。实际上，在 OpenGL 应用程序中 RC 与 DC 的工作方式是相同的。对于 GDI，DC 存储了各种图形绘图操作的状态变量，如当前笔刷、当前颜色等，RC 也与此类似。OpenGL 是一种状态机制，RC 中就存储了 OpenGL 的一系列状态命令。

3. 像素格式

像素格式决定了像素颜色的设置与 DC 和 RC 相关联的辅助缓冲区像素格式的设置，其内容是 OpenGL 调用与 Windows 绘制操作之间的中间转换设置。因此要在 Windows 环境下实现 OpenGL 的函数功能，就必须先设置像素格式。OpenGL for Windows 提供了一系列处理像素格式的函数，这些函数为：

```
ChoosePixelFormat();
SetPixelFormat();
GetPixelFormat();
DescribePixelFormat();
```

2.1.3 像素格式的结构和参数设置

2.1.2 节已经说明了 Windows 下进行 OpenGL 编程必须设置像素格式，本节中将介绍像素格式的结构以及设置像素格式的方法。像素格式包含了一系列的 OpenGL 的绘制属性，这些属性包括：RGBA 模式还是颜色索引表模式，单缓存还是双缓存，以及颜色位数、深度缓存和模板缓存等像素格式。像素格式用一个结构来表示，这个结构包含 26 个属性信息，其形式为：

```
typedef struct tagPIXELFORMATDESCRIPTOR { // pfd
WORD nSize;
WORD nVersion;
DWORD dwFlags;
```

```
BYTE iPixelType;
BYTE cColorBits;
BYTE cRedBits;
BYTE cRedShift;
BYTE cGreenBits;
BYTE cGreenShift;
BYTE cBlueBits;
BYTE cBlueShift;
BYTE cAlphaBits;
BYTE cAlphaShift;
BYTE cAccumBits;
BYTE cAccumRedBits;
BYTE cAccumGreenBits;
BYTE cAccumBlueBits;
BYTE cAccumAlphaBits;
BYTE cDepthBits;
BYTE cStencilBits;
BYTE cAuxBuffers;
BYTE iLayerType;
BYTE bReserved;
DWORD dwLayerMask;
DWORD dwVisibleMask;
DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

像素格式中参数的具体含义可参见有关文章，该像素格式结构的参数设置常用方式示例如下：

```
PIXELFORMATDESCRIPTOR pfd = {
sizeof(PIXELFORMATDESCRIPTOR), //结构大小
1, //结构版本号
PFD_SUPPORT_OPENGL | //支持OpenGL
PFD_DRAW_TO_WINDOW | //绘制到Windows
PFD_DOUBLEBUFFER, //支持双缓冲区
PFD_TYPE_RGBA, //颜色模式
16, //颜色位数
0, 0, 0, 0, 0, 0, //颜色位 可忽略
0, //alpha缓存
0, //alpha位数 可忽略
0, //累积缓存
0, 0, 0, 0, //累积缓存位数 可忽略
```

```
16, //深度缓存位数
0, //模板缓存
0, //辅助缓存
PFD_MAIN_PLANE, //主层类型
0, //保留数
0, 0, 0, //不设置层，可见性及毁损掩模
}
```

进行像素格式的设置后即可调用该格式，具体编程方法在 2.2 节中介绍。

2.2 Windows 环境下 OpenGL 编程方法

在 Windows 环境下进行 OpenGL 的编程时，需要对 Windows 窗口属性进行专门的设置。设置方式通常有多种，例如，很多程序都用辅助库 glaux 的方法，也可以用 OpenGL 的实用函数库 glut 的方法。本章重点介绍直接使用 GDI 和 WGL 函数创建 OpenGL 窗口的方法。本节通过一个典型的示例分四部分介绍 Windows 下 OpenGL 的编程方法。

示例源程序：

- 建立 GDI 窗口。
- 建立用于 OpenGL 绘制的窗口。
- 使用 OpenGL 函数绘制图形。
- 释放 DC 和 RC。

2.2.1 示例源程序

本小节用一个应用了 Windows 窗口的 OpenGL 源程序进行示范说明：

```
#include <windows.h>
#include <GL/gl.h>
char *className = "OpenGL";
char *windowName = "OpenGL Cube";
int winX = 0, winY = 0;
int winWidth = 300, winHeight = 300;
HDC hDC;
HGLRC hGLRC;
HPALETTE hPalette;
void init(void)
{
  /* 设置视图投影*/
  glMatrixMode(GL_PROJECTION);
  glFrustum(-0.5F, 0.5F, -0.5F, 0.5F, 1.0F, 3.0F);
```

```
    /* 视区定位*/
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.0F, 0.0F, -2.0F);
    /* 物体定位*/
    glRotatef(30.0F, 1.0F, 0.0F, 0.0F);
    glRotatef(30.0F, 0.0F, 1.0F, 0.0F);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}
void redraw(void)
{
    /* 清除颜色和深度缓冲区*/
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    /* 画立方体的 6 个面*/
    glBegin(GL_QUADS);
    glNormal3f( 0.0F, 0.0F, 1.0F);
    glVertex3f( 0.5F, 0.5F, 0.5F);
    glVertex3f(-0.5F, 0.5F, 0.5F);
    glVertex3f(-0.5F, -0.5F, 0.5F);
    glVertex3f( 0.5F, -0.5F, 0.5F);
    glNormal3f( 0.0F, 0.0F, -1.0F);
    glVertex3f(-0.5F, -0.5F, -0.5F);
    glVertex3f(-0.5F, 0.5F, -0.5F);
    glVertex3f( 0.5F, 0.5F, -0.5F);
    glVertex3f( 0.5F, -0.5F, -0.5F);
    glNormal3f( 0.0F, 1.0F, 0.0F);
    glVertex3f( 0.5F, 0.5F, 0.5F);
    glVertex3f( 0.5F, 0.5F, -0.5F);
    glVertex3f(-0.5F, 0.5F, -0.5F);
    glVertex3f(-0.5F, 0.5F, 0.5F);
    glNormal3f( 0.0F, -1.0F, 0.0F);
    glVertex3f(-0.5F, -0.5F, -0.5F);
    glVertex3f( 0.5F, -0.5F, -0.5F);
    glVertex3f( 0.5F, -0.5F, 0.5F);
    glVertex3f(-0.5F, -0.5F, 0.5F);
    glNormal3f( 1.0F, 0.0F, 0.0F);
    glVertex3f( 0.5F, 0.5F, 0.5F);
    glVertex3f( 0.5F, -0.5F, 0.5F);
```

```
    glVertex3f( 0.5F, -0.5F, -0.5F);
    glVertex3f( 0.5F,  0.5F, -0.5F);
    glNormal3f(-1.0F,  0.0F,  0.0F);
    glVertex3f(-0.5F, -0.5F, -0.5F);
    glVertex3f(-0.5F, -0.5F,  0.5F);
    glVertex3f(-0.5F,  0.5F,  0.5F);
    glVertex3f(-0.5F,  0.5F, -0.5F);
  glEnd();
  SwapBuffers(hDC);
}
void resize(void)
{
  /* 设置视区，使之覆盖整个窗口 */
  glViewport(0, 0, winWidth, winHeight);
}
void setupPixelFormat(HDC hDC)
{
  PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR), /* 结构的大小 */
    1, /* 版本 */
    PFD_SUPPORT_OPENGL |
    PFD_DRAW_TO_WINDOW |
    PFD_DOUBLEBUFFER, /* 支持双缓存 */
    PFD_TYPE_RGBA, /* 颜色模式 */
    16, /* 颜色深度 */
    0, 0, 0, 0, 0, 0, /* 颜色位（忽略） */
    0, /* 没有非透明度缓存 */
    0, /* 忽略非透明度位 */
    0, /* 没有累加缓存 */
    0, 0, 0, 0, /* 累加位（忽略） */
    16, /* 深度缓存 */
    0, /* 没有模块缓存 */
    0, /* 没有辅助缓存 */
    PFD_MAIN_PLANE, /* 主层 */
    0, /* 保留 */
    0, 0, 0, /* 不设置层，可见性和损毁掩模 */
  };
  int pixelFormat;
  pixelFormat = ChoosePixelFormat(hDC, &pfd);
```

```
  if (pixelFormat == 0) {
    MessageBox(WindowFromDC(hDC), "ChoosePixelFormat failed.",
             "Error", MB_ICONERROR | MB_OK);
    exit(1);
  }
  if (SetPixelFormat(hDC, pixelFormat, &pfd) != TRUE) {
    MessageBox(WindowFromDC(hDC), "SetPixelFormat failed.",
             "Error", MB_ICONERROR | MB_OK);
    exit(1);
  }
}
void setupPalette(HDC hDC)
{
  int pixelFormat = GetPixelFormat(hDC);
  PIXELFORMATDESCRIPTOR pfd;
  LOGPALETTE* pPal;
  int paletteSize;
  DescribePixelFormat(hDC, pixelFormat,
                sizeof(PIXELFORMATDESCRIPTOR), &pfd);
  if (pfd.dwFlags & PFD_NEED_PALETTE) {
    paletteSize = 1 << pfd.cColorBits;
  } else {
    return;
  }
  pPal = (LOGPALETTE*)
  malloc(sizeof(LOGPALETTE) + paletteSize *
    sizeof(PALETTEENTRY));
    pPal->palVersion = 0x300;
    pPal->palNumEntries = paletteSize;
    /* 建一个简单的 RGB 调色板 */
  {
    int redMask = (1 << pfd.cRedBits) - 1;
    int greenMask = (1 << pfd.cGreenBits) - 1;
    int blueMask = (1 << pfd.cBlueBits) - 1;
    int i;
    for (i=0; i<paletteSize; ++i) {
      pPal->palPalEntry[i]-peRed =
         (((i >> pfd.cRedShift) & redMask) * 255) / redMask;
      pPal->palPalEntry[i]-peGreen =
```

```
(((i >> pfd.cGreenShift) & greenMask) * 255) / greenMask;
pPal->palPalEntry[i]-peBlue =
(((i >> pfd.cBlueShift) & blueMask) * 255) / blueMask;
pPal->palPalEntry[i]-peFlags = 0;
    }
  }
  hPalette = CreatePalette(pPal);
  free(pPal);
  if (hPalette) {
    SelectPalette(hDC, hPalette, FALSE);
    RealizePalette(hDC);
  }
}
LRESULT APIENTRY WndProc(
HWND hWnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    case WM_CREATE:
      /* 为 OpenGL 绘制进行初始化 */
      hDC = GetDC(hWnd);
      setupPixelFormat(hDC);
      setupPalette(hDC);
      hGLRC = wglCreateContext(hDC);
      wglMakeCurrent(hDC, hGLRC);
      init();
      return 0;
    case WM_DESTROY:
      /* 结束 OpenGL 绘制 */
      if (hGLRC) {
        wglMakeCurrent(NULL, NULL);
        wglDeleteContext(hGLRC);
      }
      if (hPalette) {
        DeleteObject(hPalette);
      }
      ReleaseDC(hWnd, hDC);
      PostQuitMessage(0);
      return 0;
```

```
case WM_SIZE:
  /* 跟踪窗口大小的改变 */
  if (hGLRC) {
    winWidth = (int) LOWORD(lParam);
    winHeight = (int) HIWORD(lParam);
    resize();
    return 0;
  }
case WM_PALETTECHANGED:
  /* 如果它不是当前窗口，则实现调色板与 hDC 的对应*/
  if (hGLRC && hPalette && (HWND) wParam != hWnd) {
  UnrealizeObject(hPalette);
  SelectPalette(hDC, hPalette, FALSE);
  RealizePalette(hDC);
  redraw();
  break;
}
break;
case WM_QUERYNEWPALETTE:
  /* 如果是当前窗口，实现调色板与 hDC 的对应 */
  if (hGLRC && hPalette) {
  UnrealizeObject(hPalette);
  SelectPalette(hDC, hPalette, FALSE);
  RealizePalette(hDC);
  redraw();
  return TRUE;
}
break;
case WM_PAINT:
{
  PAINTSTRUCT ps;
  BeginPaint(hWnd, &ps);
  if (hGLRC) {
    redraw();
  }
  EndPaint(hWnd, &ps);
  return 0;
}
break;
```

```
        case WM_CHAR:
          /* 处理键盘输入 */
          switch ((int)wParam) {
            case VK_ESCAPE:
              DestroyWindow(hWnd);
              return 0;
            default:
              break;
          }
          break;
        default:
        break;
      }
      return DefWindowProc(hWnd, message, wParam, lParam);
    }
    int APIENTRY WinMain(
    HINSTANCE hCurrentInst,
    HINSTANCE hPreviousInst,
    LPSTR lpszCmdLine, int nCmdShow)
    {
      WNDCLASS wndClass;
      HWND hWnd;
      MSG msg;
      /* 注册窗口类 */
      wndClass.style = CS_HREDRAW | CS_VREDRAW;
      wndClass.lpfnWndProc = WndProc;
      wndClass.cbClsExtra = 0;
      wndClass.cbWndExtra = 0;
      wndClass.hInstance = hCurrentInst;
      wndClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
      wndClass.hbrBackground = GetStockObject(BLACK_BRUSH);
      wndClass.lpszMenuName = NULL;
      wndClass.lpszClassName = className;
      RegisterClass(&wndClass);
      /* 创建窗口 */
      hWnd = CreateWindow(
        className, windowName,
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
```

```
        winX,  winY,  winWidth,  winHeight,
        NULL,  NULL,  hCurrentInst,  NULL);
    /* 显示窗口 */
    ShowWindow(hWnd,  nCmdShow);
    UpdateWindow(hWnd);
    /* 处理消息 */
    while (GetMessage(&msg,  NULL,  0,  0) == TRUE) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return msg.wParam;
```

2.2.2　建立 GDI 窗口

任何一个 Windows 应用程序都必须创建 Windows 应用窗口。建立 Windows 应用窗口的方法，与普通的 Windows 应用程序的建立方法相同，在程序中通过调用 WinMain()函数来实现。在 WinMain()中，定义和注册一个窗口类 winClass，这个类结构定义了一个父窗口，其他菜单之类的子窗口都是基于该窗口的。在示例程序中进行如下定义：

```
WNDCLASS wndClass;
HWND hWnd;
MSG msg;
/* register window class */
wndClass.style = CS_HREDRAW | CS_VREDRAW;
wndClass.lpfnWndProc = WndProc;
wndClass.cbClsExtra = 0;
wndClass.cbWndExtra = 0;
wndClass.hInstance = hCurrentInst;
wndClass.hIcon = LoadIcon(NULL,  IDI_APPLICATION);
wndClass.hCursor = LoadCursor(NULL,  IDC_ARROW);
wndClass.hbrBackground = GetStockObject(BLACK_BRUSH);
wndClass.lpszMenuName = NULL;
wndClass.lpszClassName = className;
RegisterClass(&wndClass);
/* 创建窗口 */
hWnd = CreateWindow(
className,  windowName,
WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
winX,  winY,  winWidth,  winHeight,
NULL,  NULL,  hCurrentInst,  NULL);
```

```
/* 显示窗口 */
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
```

在 WinMain()的最后通过一个循环处理所有消息事件，这些消息事件是在 WinPro()函数中定义的。

```
while (GetMessage(&msg, NULL, 0, 0) == TRUE) {
TranslateMessage(&msg);
DispatchMessage(&msg);
while (GetMessage(&msg, NULL, 0, 0) == TRUE) {
TranslateMessage(&msg);
DispatchMessage(&msg);
}
return msg.wParam;
```

2.2.3 建立用于 OpenGL 绘制的窗口

在前面已经说明，在Windows环境下建立OpenGL绘制窗口是通过建立设备描述表(DC)和绘制描述表(RC)来实现的，而 RC 与 DC 是连接在一起的，因此为了建立 RC，首先要建立 DC。下面分两步来介绍这一过程。

1. 创建设备描述表

在 WinProc()函数中，用 WM_CREATE 命令获得和创建 DC。

```
hDC = GetDC(hWnd);
```

为创建专用于 OpenGL 绘制的 DC，必须定义与普通 DC 不同的像素格式。关于像素格式的具体说明已在前面章节中讨论，在本示例中定义一个函数以实现像素格式的定义：

```
setupPixelFormat(hDC);
```

在该函数中首先定义像素格式结构：

```
PIXELFORMATDESCRIPTOR(pfd)
```

这种结构定义了窗口的属性：

```
PIXELFORMATDESCRIPTOR pfd = {
sizeof(PIXELFORMATDESCRIPTOR), /* size of this pfd */
1, /* 版本号 */
PFD_SUPPORT_OPENGL |
PFD_DRAW_TO_WINDOW | /* 窗口类型 */
PFD_DOUBLEBUFFER,
PFD_TYPE_RGBA, /* RGBA 颜色模式 */
```

参数 PFD_DOUBLEBUFFER 表明使用双缓存。

在结构 pfd 定义完毕后，调用 ChoosePixelFormat()确定 pfd 结构是否存在。如果返回值为 TRUE，则调用函数 SetPixelFormat()为 DC 设置像素格式。在示例程序中进行了这样

的定义：

```
int pixelFormat;
pixelFormat = ChoosePixelFormat(hDC, &pfd);
if (SetPixelFormat(hDC, pixelFormat, &pfd) != TRUE)
{…
}
```

到此为止，DC 已经创建完毕。但在本示例中加入了创建调色板函数：

```
setupPalette(hDC);
```

使用该函数的目的是，即使在图形显示驱动程序支持的颜色数为 256 色的条件下，窗口仍能很好地表现 OpenGL 图形色彩。是否使用调色板在像素格式结构的 dwFlag 中定义，参数为 PFD_NEED_PALETTE。在本示例中实际上已经忽略了这一过程。

2. 创建绘制描述表

Windows 环境下创建绘制描述表的方法是，首先建立按照当前像素格式的绘制文本，然后确认绘制文本处于当前状态。OpenGL 绘图命令不直接与视口交互，而是总是在绘制文本上绘图。具体函数调用方法是：

```
hGLRC = wglCreatContext (hDC);
wglMakeCurrent(hDC, hGLRC);
```

其中，wglCreateContext()函数建立一个用于 OpenGL 的绘制文本。如果 wglCreateContext()调用成功，则函数返回一个绘制描述表的句柄(hGLRC)；如果失败，则函数返回 NULL 值。

图形显示硬件支持多种像素格式，这可以通过驱动程序支持的颜色数来体现，如在系统桌面的[属性]项的[设置]选项卡中看到的颜色数，包括 8 位、16 位或 256 色。因此，Windows 应用程序也需要建立与当前像素格式相对应的 DC，然后把 RC 与 DC 连接在一起。这项工作就是通过调用 wglCreateContext()函数实现的。

建立绘制描述表后，就要把 RC 与 DC 连接在一起，这通过调用函数 wglMakeCurrent()来实现。如果 wglMakeCurrent()调用成功，返回值为 TRUE，否则返回值为 FALSE。通过调用 wglMakeCurrent()，应用程序可以随时连接到一个不同的 RC。另外，如果调用 wglMakeCurrent()，把 RC 与 NULL 连接在一起，则相当于释放了 RC。

2.2.4 用 OpenGL 绘图

创建绘制描述表后，现在就可以用 OpenGL 在 RC 中绘制图形。在本示例的函数 Redraw()就是在 RC 中用 OpenGL 绘制一个彩色立方体。

```
void redraw(void)
{
  /* 清除颜色和深度缓冲区 */
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  /* 绘制立方体的 6 个面 */
```

```
    glBegin(GL_QUADS);
    glNormal3f( 0.0F,  0.0F,  1.0F);
    glVertex3f( 0.5F,  0.5F,  0.5F);
    glVertex3f(-0.5F,  0.5F,  0.5F);
    glVertex3f(-0.5F, -0.5F,  0.5F);
    glVertex3f( 0.5F, -0.5F,  0.5F);
    glNormal3f( 0.0F,  0.0F, -1.0F);
    glVertex3f(-0.5F, -0.5F, -0.5F);
    glVertex3f(-0.5F,  0.5F, -0.5F);
    glVertex3f( 0.5F,  0.5F, -0.5F);
    glVertex3f( 0.5F, -0.5F, -0.5F);
    glNormal3f( 0.0F,  1.0F,  0.0F);
    glVertex3f( 0.5F,  0.5F,  0.5F);
    glVertex3f( 0.5F,  0.5F, -0.5F);
    glVertex3f(-0.5F,  0.5F, -0.5F);
    glVertex3f(-0.5F,  0.5F,  0.5F);
    glNormal3f( 0.0F, -1.0F,  0.0F);
    glVertex3f(-0.5F, -0.5F, -0.5F);
    glVertex3f( 0.5F, -0.5F, -0.5F);
    glVertex3f( 0.5F, -0.5F,  0.5F);
    glVertex3f(-0.5F, -0.5F,  0.5F);
    glNormal3f( 1.0F,  0.0F,  0.0F);
    glVertex3f( 0.5F,  0.5F,  0.5F);
    glVertex3f( 0.5F, -0.5F,  0.5F);
    glVertex3f( 0.5F, -0.5F, -0.5F);
    glVertex3f( 0.5F,  0.5F, -0.5F);
    glNormal3f(-1.0F,  0.0F,  0.0F);
    glVertex3f(-0.5F, -0.5F, -0.5F);
    glVertex3f(-0.5F, -0.5F,  0.5F);
    glVertex3f(-0.5F,  0.5F,  0.5F);
    glVertex3f(-0.5F,  0.5F, -0.5F);
    glEnd();
    SwapBuffers(hDC);
}
```

注意： 该函数是在 WndProc()的 WM_PAINT 消息处理过程中，即当应用程序获得消息后，才调用 Redraw()函数绘制立方体。

2.2.5 释放设备描述表和绘制描述表

在 WndProc()中的 WM_DESTROY 消息用于释放设备描述表和绘制描述表。具体方法为：

- 释放绘制描述表

```
wglMakeCurrent(NULL, NULL);
```

- 删除绘制描述表

```
wglDeleteContext(hGLRC);
```

- 释放设备描述表

```
ReleaseDC(hWin, hDC);
```

在这个消息中同样可以处理应用程序的其他关闭操作。

2.3 利用 Visual C++ 6.0 的 AppWizard 编写 OpenGL 应用程序

通过 2.2 节的讲解，用户应该知道，在 Windows 下用 GDI 作图必须通过设备描述表(Device Context，简写为 DC)调用相应的函数。用 OpenGL 作图也类似，OpenGL 函数是通过绘制描述表(Rendering Context，简写 RC)完成三维图形的绘制。Windows 下的窗口和设备描述表支持“位图格式”(Pixel Fomet)属性，和 RC 有着位图结构上的一致。只要在创建RC时与一个DC建立联系(RC也只能通过已经建立了位图格式的DC来创建)，OpenGL 的函数就可以通过 RC 对应的 DC 画到相应的显示设备上。其中，还有以下几方面需要注意：

- 一个线程只能拥有一个 RC

用户如果在一个线程内对不同设备作图，只能通过更换与 RC 对应的 DC 来完成，而 RC 在线程中保持不变(当然，删除旧的 RC 后再创建新的 RC 是可以的)。与此对应，一个 RC 也只能属于一个线程，不能被不同线程同时共享。

- DC 位图格式与窗口的位图格式是一一对应的

设定 DC 位图格式等于设定了相应窗口的位图格式，DC 和窗口的位图格式一旦确定就不能再改变。这一点只能期望在以后的 Windows 版本中有所改进。

- 任何时刻只能利用一个 DC

虽然一个 RC 可以更换 DC，在任何时刻只能利用一个 DC(该 DC 称为 RC 的当前 DC)，但由于一个窗口可以让多个 DC 作图，从而可以让多个线程利用多个 RC 在该窗口上执行 OpenGL 操作。

- 对 GDI 函数使用的一些限制

目前 Windows 环境下的 OpenGL 版本对 OpenGL 和 GDI 在同一个 DC 上作图有一定的限制。当使用双缓存用 OpenGL 产生动画时，不能使用 GDI 函数向该 DC 作图。

- 尽量不要使用标准 C 编写 OpenGL 程序

建议不用标准 C 在 Windows 下编写 OpenGL 程序。因为这样的程序虽然具有跨平台的可移植性(比如很多 SGI 的程序)，但是它们不能利用 Windows 操作系统的很多特性，实用价值不大。

下面介绍如何使用 Visual C++ 6.0 提供的 App Wizard 设置编程环境。

2.3.1 创建一个新工程

从 File 菜单中选择 New 菜单项，在弹出的的对话框中选择 Projects 标签(如图 2.1 所示)。然后单击 MFC AppWizard 选项，再给应用程序指定目录和名字，如工程名为 OGLTest。

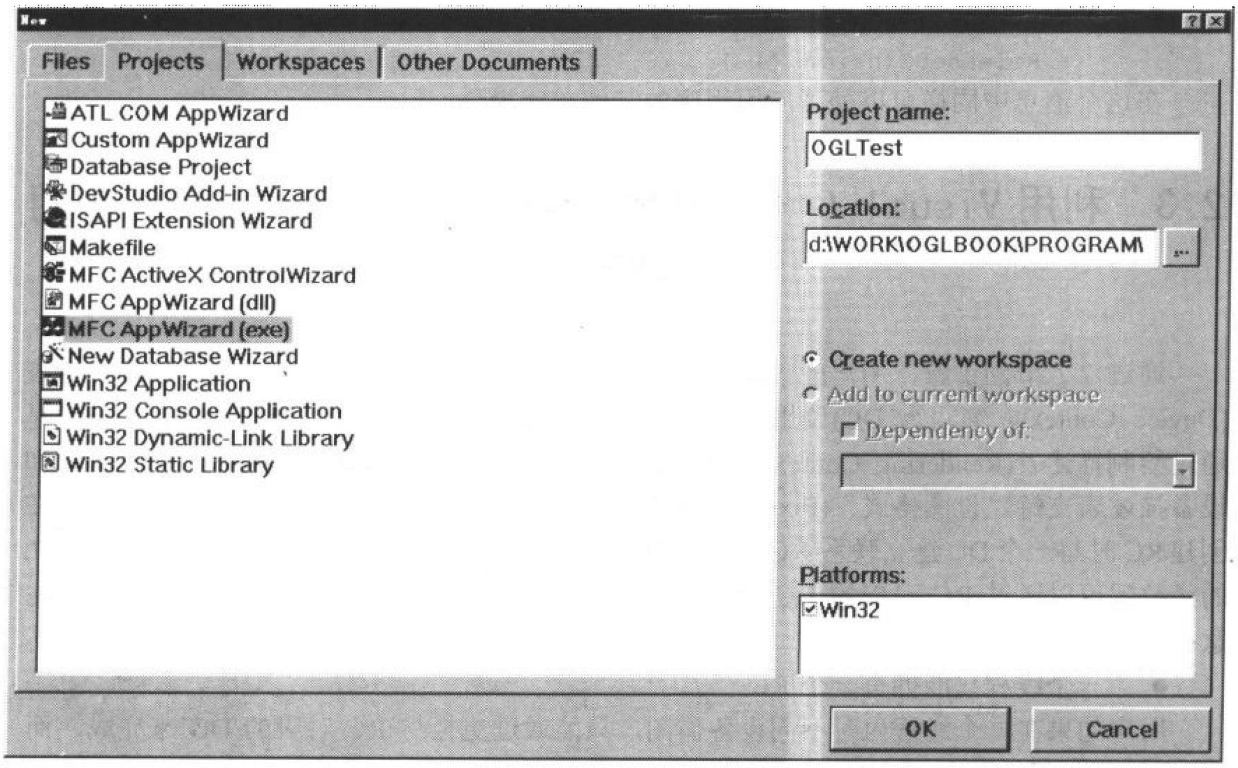

图2.1 选择应用程序类型

单击 OK 按钮，确认由 Visual C++ 6.0 创建一个新工程，接着 Visual C++ 6.0 将显示如图 2.2 所示的对话框，用以选择用单文档还是多文档界面。

如果只需要学习 OpenGL 的一些常用函数、命令及各种概念时，通常可以选择建立单文档界面，当然根据用户需要有时需要使用多文档界面方式，我们在后面的章节中将给出一个使用多文档界面的例子。接下来的对话框用来设置支持何种数据库的信息，由于本书不涉及数据库的内容，因此每个程序均选择不支持数据库，如图 2.3 所示。

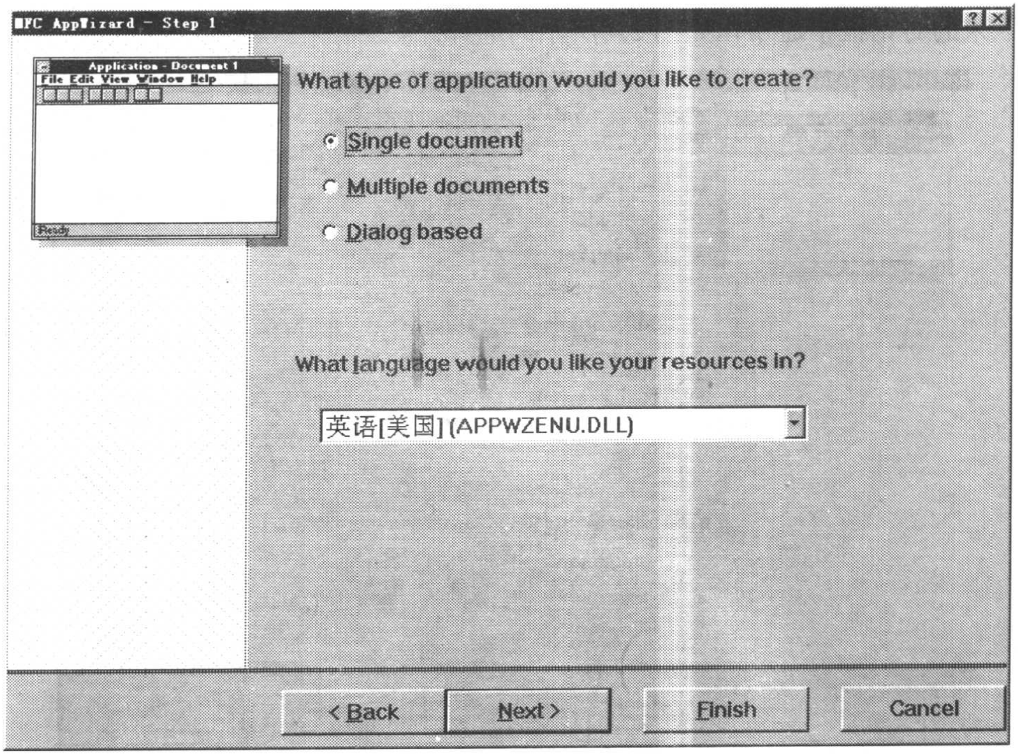

图2.2　选择单文档或多文档

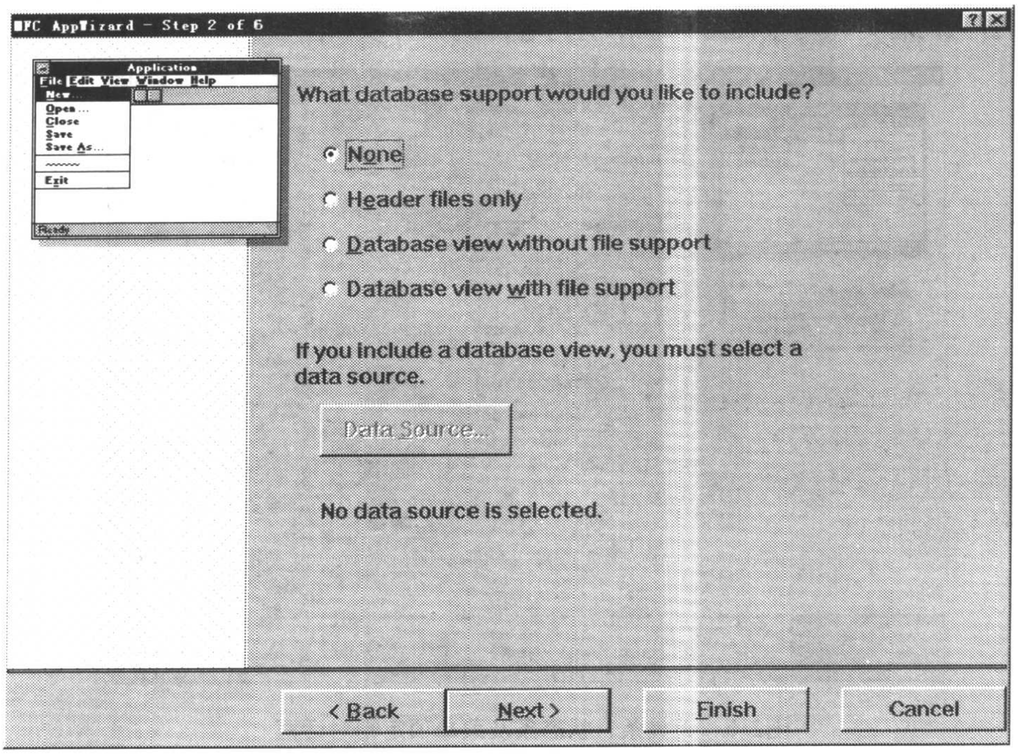

图2.3　选择是否支持数据库

接下来的对话框中通常作如图 2.4 所示的选择。

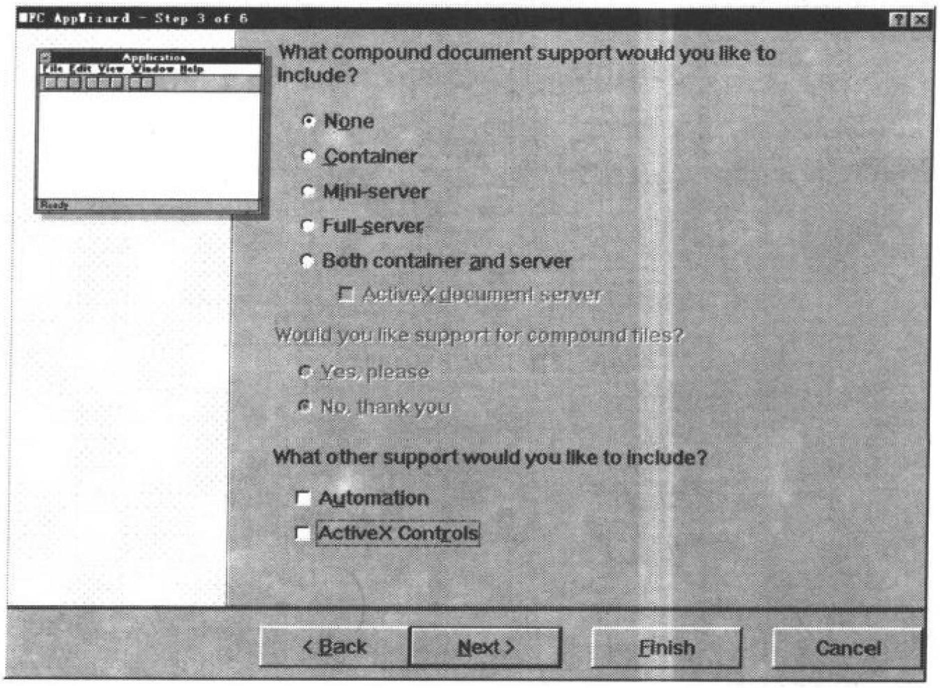

图2.4　选择文档的 OLE 或 ActiveX 控件的支持

根据各人的喜好填写下面对话框中的内容，例如可以如图 2.5 所示填写。

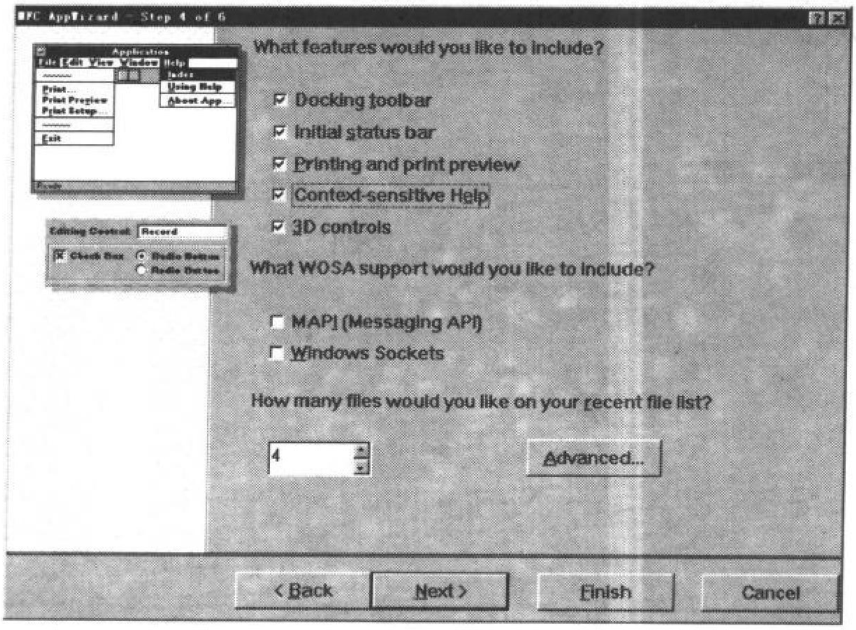

图2.5　工具条、状态条及菜单项定制对话框

选择好以后单击 Finish 按钮，一个新的工程框架就建立了。

2.3.2　设置编译环境

设置编译环境主要包括设置包含文件和设置库文件路径，从 Tools 菜单中选择 Options 菜单项，在弹出的的对话框(如图 2.6 所示)中选择 Directories 标签，再设置包含文件及库文件所在目录。在默认情况下，两者的目录已正确设置，即 Visual C++ 6.0 目录的 include 子目录和 lib 子目录。

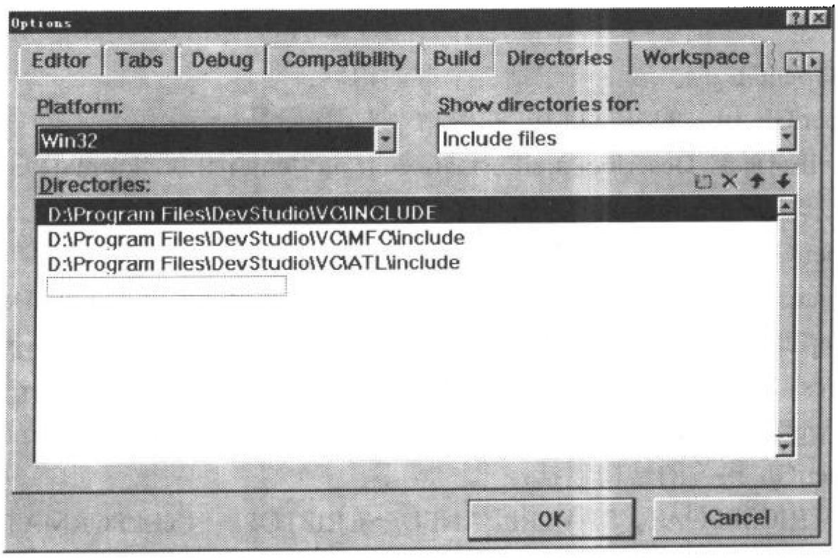

图2.6　设置包含文件及库文件对话框

有时需要增加一些自己定义的包含文件，此时可以双击 Directories 列表框中的空白处，在其中输入希望查询的包含路径即可。建议将自己建立的包含文件放在主文件所在的同一个目录下，这样就不必在此增加包含路径，因为 Visual C++ 6.0 会自动对当前路径进行搜索。

2.3.3　加入 OpenGL 库

从 Project 菜单中选择 Settings 菜单项，在弹出的对话框中单击 Link 标签，在 Object/Library Module 文本框中加入 opengl32.lib glu32.lib glaux.lib 即可(Windows 95 下相应的库文件为 opengl.lib，glu.lib 和 glaux.lib)。

完成了以上步骤之后，一个在 Windows 环境下进行 OpenGL 编程开发的框架便形成了，接着进入具体的编程阶段。查看 WorkSpace 用户会发现 Visual C++ 6.0 的 App Wizard 已经为用户生成了多个类，其中包括：App 类、Mainframe 类、View 类和 Document 类等。其中与 OpenGL 编程最相关的是 View 类，主要的 OpenGL 显示任务都将在其中完成。当然，对其他类的编程管理也是必不可少的，例如可以在 Mainframe 类中设置窗口的大小、窗口的组成，在 Document 类中可以读入 OpenGL 所需的外部数据等。

本节讲述了 Visual C++ 6.0 中进行 OpenGL 编程的一些基本知识和内容。

第 3 章　Win32 环境下 OpenGL 相关函数及结构

Win32 对 OpenGL 的扩展主要包括 3 个方面内容：

- 与 OpenGL 相关的 Win32 函数共有 6 个，它们是：

ChoosePixelFormat，DescribePixelFormat，GetEnhMetaFilePixelFormat，GetPixelFormat，SetPixelFormat 和 SwapBuffers。

- 与 OpenGL 相关的 wgl 函数包括：

wglCopyContext，wglCreateContext，wglCreateLayerContext，wglDeleteContext，wglDescribeLayerPlane，wglGetCurrentContext，wglGetCurrentDC，wglGetLayerPaletteEntries，wglGetProcAddress，wglMakeCurrent，wglRealizeLayerPalette，wglSetLayerPaletteEntries，wglShareLists，wglSwapLayerBuffers，wglUseFontBitmaps 和 wglUseFontOutlines。

- 与 OpenGL 相关的结构包括：

GLYPHMETRICSFLOAT，LAYERPLANEDESCRIPTOR，PIXELFORMATDESCRIPTOR 和 POINTFLOAT。

下面分别对上述内容进行详细讲解。

3.1　与 OpenGL 相关的 Win32 函数

与 OpenGL 相关的 Win32 函数共有 6 个，下面逐个加以详细讲解。

3.1.1　ChoosePixelFormat 函数

该函数的原型如下所示：

```
int ChoosePixelFormat( HDC hdc, CONST PIXELFORMATDESCRIPTOR * ppfd );
```

其功能是：使一个设备描述表支持的合适像素格式与一个给定的像素格式定义相匹配。它有 2 个参数：hdc 和 ppfd，其中，hdc 给定与 ppfd 指向的像素格式描述符相匹配的设备描述表，ppfd 用于指定所需像素格式的 PIXELFORMATDESCRIPTOR 结构指针。PIXELFORMATDESCRIPTOR 结构由以下成分构成：

- nSize：给定 PIXELFORMATDESCRIPTOR 数据结构的大小，即 sizeof (PIXELFORMATDESCRIPTOR)。
- nVersion：PIXELFORMATDESCRIPTOR 数据结构的版本号，这个成分设为 1。
- dwFlags：一系列定义像素缓冲区属性的位标识。可以用逐位进行逻辑“或”运

算把下列标识组合起来。设定下列任一标识时，ChoosePixelFormat 函数将把肯有那个标识或标识集合的像素格式匹配起来，否则将在像素格式中忽略这个标识。

```
PFD_DRAW_TO_WINDOW
PFD_DRAW_TO_BITMAP
PFD_SUPPORT_GDI
PFD_SUPPORT_OPENGL
```

若设定 PFD_DOUBLEBUFFER 和 PFD_STEREO，ChoosePixelFormat 函数将把具有那个标识或标识集合的像素格式匹配起来，否则将在像素格式中忽略这个标识。

若设定 PFD_DOUBLEBUFFER_DONTCARE，函数将忽略像素格式 PFD_ DOUBLEBUFFER 和 PFD_STEREO

- iPixelType 指定像素格式的如下类型：
  PFD_TYPE_RGBA，PFD_TYPE_COLORINDEX
- cColorBits：大于等于零
- cRedBits：不使用
- cRedShift：不使用
- cGreenBits：不使用
- cGreenShift：不使用
- cBlueBits：不使用
- cBlueShift：不使用
- cAlphaBits：大于等于零
- cAlphaShift：不使用
- cAccumBits：大于等于零
- cAccumRedBits：不使用
- cAccumGreenBits：不使用
- cAccumBlueBits：不使用
- cAccumAlphaBits：不使用
- cDepthBits：大于等于零
- cStencilBits：大于等于零
- cAuxBuffers：大于等于零
- iLayerType：大于等于零
  iLayerType 指定下列层类型值之一：

PFD_MAIN_PLANE，PFD_OVERLAY_PLANE 和 PFD_UNDERLAY_PLANE。

- bReserved：不使用
- dwLayerMask：不使用
- dwVisibleMask：不使用
- dwDamageMask：不使用

该函数有一个整型返回值，当调用函数成功时，返回值与像素格式最能匹配的一个像素格式指数；若调用函数失败，返回值为 0。可以调用 GetLastError 得到进一步的出错信

息。

注意：一定要使 ChoosePixelFormat 函数匹配的像素格式满足用户要求。如果需要有 24 位 RGB 彩色缓冲区的像素格式，但是设备只能提供一个 8 位 RGB 的缓冲区，那么，函数返回的像素格式也只能是一个 8 位 RGB 缓冲区的像素格式。

下面举例说明该函数的使用：

```
PIXELFORMATDESCRIPTOR pfd =
sizeof(PIXELFORMATDESCRIPTOR),   //  该结构的大小
1, // version number
PFD_DRAW_TO_WINDOW |    // 支持窗口中进行绘制
PFD_SUPPORT_OPENGL |    // 支持 OpenGL
PFD_DOUBLEBUFFER,    // 双缓存
PFD_TYPE_RGBA,  // RGBA 颜色模式
24, // 24 位颜色深度
0,  0,  0,  0,  0,  0,   // 忽略颜色位
0,  // 没有非透明缓存
0,  // 忽略移位位
0,  // 没有累加缓存
0,  0,  0,  0, // 忽略累加位
32, // 32 位深度缓存
0,  // 没有模板缓存
0,  // 没有辅助缓存
PFD_MAIN_PLANE, // 主层
0,  // 保留
0,  0,  0// 忽略层，可见性及损毁掩模
};
HDC  hdc;
int  iPixelFormat;
iPixelFormat = ChoosePixelFormat(hdc,  &pfd);
```

相关函数：DescribePixelFormat，GetPixelFormat 和 SetPixelFormat。

3.1.2 DescribePixelFormat 函数

该函数的功能是提供与 hdc 参数相关的。由 iPixelFormat 标识的像素格式信息，并由像素格式数据设定 ppfd 指向的 PIXELFORMATDESCRIPTOR 结构。函数原型如下：

```
int DescribePixelFormat( HDC hdc,  int iPixelFormat,  UINT nBytes,
LPPIXELFORMATDESCRIPTOR ppfd );
```

该函数 4 个参数的含义分别描述如下：

- hdc：用于定义设备描述表。
- iPixelFormat：像素格式的指数。

- nBytes：由 ppfd 指向的结构字节数。这个值设为 sizeof 当 (PIXELFORMATDESCRIPTOR)，DescribePixelFormat 只能在结构中存 nBytes 个字节。
- ppfd：指向 PIXELFORMATDESCRIPTOR 结构的指针。把像素格式数据赋给这个结构，由结构中的 nSize 成分给出函数拷贝到结构中的字节数。若 ppfd 是 NULL，函数并不向结构中写数据，在用户只想得到设备描述表能提供最多像素格式的数目时该参数非常有用。
- 返回值：若调用函数成功，返回值是设备描述表能提供的最大像素格式数目，并按给定的像素格式，设定由 ppfd 指向的 PIXELFORMATDESCRIPTOR 结构成分；若调用函数失败，返回值为 0。

下面程序给出该函数的用法：

```
PIXELFORMATDESCRIPTOR  pfd;
HDC  hdc;
int  iPixelFormat;

iPixelFormat = 1;

// 得到设备描述表第一个象素格式的详细信息//
DescribePixelFormat(hdc,  iPixelFormat,
sizeof(PIXELFORMATDESCRIPTOR),  &pfd);
```

3.1.3 GetPixelFormat 函数

该函数的功能是提供指定设备描述表中当前像素格式指数。函数的原型如下：

```
int GetPixelFormat( HDC hdc );
```

参数 hdc 用于给定由函数返回当前选择的像素格式指数。

该函数有一个整型返回值，若调用函数成功，返回值是给定设备描述表的当前像素格式指数；若调用函数失败，返回值为 0。

下面程序演示了该函数的用法：

```
PIXELFORMATDESCRIPTOR  pfd;
HDC  hdc;
int  iPixelFormat;

// 得到当前像素格式索引
iPixelFormat = GetPixelFormat(hdc);

// 得到像素格式的详细描述
DescribePixelFormat(hdc,  iPixelFormat,
sizeof(PIXELFORMATDESCRIPTOR),  &pfd);
```

3.1.4 GetEnhMetaFilePixelFormat 函数

该函数的功能是给一个增强的元文件提供像素格式信息。

```
UNIT GetEnhMetaFilePixelFormat( HENHMETAFILE hemf,  DWORD cbBuffer,
CONST PIXELFORMATDESCRIPTOR * ppfd );
```

函数有 3 个参数：hemf，cbBuffer 和 ppfd，其中，hemf 参数是增强的元文件标识，cbBuffer 参数给定拷入像素格式信息的缓冲区字节数，ppfd 是指向 PIXELFORMATDESCRIPTOR 结构的指针，元文件用这个结构记录逻辑像素格式的定义。

函数有一个整型返回值，若调用函数成功，返回值是元文件像素格式的大小；若没有像素格式，返回值为 0；若出现错误或调用函数失败，返回值是 GDI_ERROR。

注意： 当增强元文件在其 ENHMETAHEADER 结构中定义一个像素格式，并且像素格式与缓冲区匹配时，像素格式信息拷贝到 ppfd。若 cbBuffer 太小，不能容纳元文件的像素格式，像素格式不会拷贝到缓冲区。上述两种情况，函数的返回值都是元文件的像素格式大小。

3.1.5 SetPixelFormat 函数

该函数的功能是给一个增强的元文件提供像素格式信息。函数原型为：

```
BOOL SetPixelFormat( HDC hdc,  int iPixelFormat,
CONST PIXELFORMATDESCRIPTOR * ppfd );
```

它有 3 个参数：hdc，iPixelFormat 和 ppfd。其中，hdc 参数是由函数赋值设定像素格式的设备描述表，iPixelFormat 参数是待赋值的像素格式指数，ppfd 同样是一个指向 PIXELFORMATDESCRIPTOR 结构的指针，这个结构不影响 SetPixelFormat 的性质。

该函数有一个布尔量返回值，若调用函数成功，返回值 TRUE；若调用函数失败，返回值为 FALSE。

注意： 若 hdc 是一个窗口，调用 SetPixelFormat 函数将改变窗口的像素格式。多次给窗口的像素格式赋值，会使窗口管理器和多线程应用软件变得复杂，这也是不允许的。窗口的像素格式一旦赋值就不能改动。

因为 wglCreateContext 函数要用一个选定的像素格式在设备上显示图形，所以在调用 wglCreateContext 函数前，应在设备描述表中选择一个像素格式。

因为 OpenGL 窗口有自己的像素格式，只有检索过 OpenGL 窗口客户区的设备描述表才允许在窗内画图。OpenGL 窗口只能用 WS_CLIPCHILDREN 和 WS_CLIPSIBLINGS 风格创建，另外，窗口类属性不能包括 CS_PARENTDC 风格。

下面程序给出该函数的用法：

```
PIXELFORMATDESCRIPTOR pfd = {
sizeof(PIXELFORMATDESCRIPTOR),    // 该结构的大小
1,  // 版本号
PFD_DRAW_TO_WINDOW |    // 支持在窗口中绘图 support window
```

```
PFD_SUPPORT_OPENGL |    // 支持 OpenGL
PFD_DOUBLEBUFFER,    // 双缓存模式
PFD_TYPE_RGBA,  // RGBA 颜色模式
24, // 24 位颜色深度
0,  0,  0,  0,  0,  0,   // 忽略颜色位
0,  // 没有非透明度缓存
0,  // 忽略移位位
0,  // 没有累加缓存
0,  0,  0,  0, // 忽略累加位
32, // 32 位深度缓存
0,  // 没有模板缓存
0,  // 没有辅助缓存
PFD_MAIN_PLANE, // 主层
0,  // 保留
0,  0,  0// 忽略层，可见性和损毁掩模
};
HDC  hdc;
int  iPixelFormat;

// 为设备描述表得到最匹配的像素格式
iPixelFormat = ChoosePixelFormat(hdc, &pfd);

// make that the pixel format of the device context
SetPixelFormat(hdc, iPixelFormat, &pfd);
```

3.1.6　SwapBuffers 函数

该函数的功能是若当前窗口引用的像素格式包括一个后台缓冲区，则交换前台和后台缓冲区。函数原型如下：

```
BOOL SwapBuffers( HDC hdc );
```

其中参数 hdc 用于给定一个设备描述表。

该函数有一个布尔量返回值，若调用函数成功，返回值为 TRUE；若调用函数失败，返回值为 FALSE。

注意：*若窗口的当前像素格式不包括后台缓冲区，这个函数不产生任何作用，在函数返回时，后台缓冲区的内容不确定。*

对于多线程程序，调用 SwapBuffers 之前，应该确保同一窗口的其他线程命令执行完毕。

3.2 与 OpenGL 相关的 wgl 函数

3.2.1 wglCreateContext 函数

该函数的功能是产生一个新的 OpenGL 绘图描述表，使之适于在 hdc 给出的设备上画图，并具有与之相同的像素格式。函数的原型如下：

```
HGLRC wglCreateContext( HDC hdc );
```

其中参数 hdc 用于给定一个设备描述表。

该函数有一个结构指针型返回值，若调用函数成功，返回值是合法的 OpenGL 绘图描述表的句柄；若调用函数失败，返回值为 NULL。

注意： 绘图描述表不同于设备描述表，在创建绘图描述表之前，要给设备描述表的像素格式赋值，有关像素格式赋值的信息，见 SetPixelFormat 函数。在使用 OpenGL 前，要创建一个绘图描述表，而在完成绘图描述表时，可用 wglDeleteContext 函数清除它。

以下程序演示 wglCreateContext 函数的用法：

```
HDChdc;
HGLRC  hglrc;

// 创建绘制描述表
hglrc = wglCreateContext (hdc);

// 使之成为调用线程的当前绘制描述表
wglMakeCurrent (hdc,  hglrc);
// 按要求调用 OpenGL 的 API
// 这时不再需要绘制描述表
// 使之成为非当前绘制描述表
wglMakeCurrent (NULL,  NULL) ;
// 删除绘制描述表
wglDeleteContext (hglrc);
```

3.2.2 wglCreateLayerContext 函数

该函数的功能是在设备描述表的指定层上创建一个新 OpenGL 绘图描述表。其原型如下所示：

```
HGLRC wglCreateLayerContext( HDC hdc,  int iLayerPlane );
```

其中，参数 hdc 给一个新的绘图描述指定设备描述表。参数 iLayerPlane 用于定义显

示文本所在的层，0 指主平面；正值标识覆盖层平面；负值代表底层平面。

覆盖层和底层平面的代号由像素格式描述符结构中的 bReserve 成分给出。

函数有一个指针型返回值，若调用函数成功，返回值是 OpenGL 绘图描述表的句柄；若调用失败，返回值是 NULL。

注意： 绘图描述表是所有 OpenGL 命令通过的端口。每个调用 OpenGL 函数的线程都要有一个当前激活的绘图描述表。与设备描述表不同，它有针对 OpenGL 的信息，而设备描述表中有 GDI 的信息。

以下程序说明如何绘制到第一覆盖层，该例中假定 hdc 的像素格式中包含了第一覆盖层平面。

```
HDC hdc;
HGLRC;
// 为第一覆盖层创制绘制描述表
hglrc = wglCreateLayerContext(hdc, 1);
// 使之成为调用线程的当前绘制描述表
wglMakeCurrent(hdc, hglrc);
// 调用 OpenGL 函数
// 这时不再需要绘制描述表
// 使之成为非当前绘制描述表
wglMakeCurrent(NULL, NULL);
// 删除绘制描述表
wglDeleteContext(hglrc);
```

3.2.3 wglcopyContext 函数

该函数的功能是把选定的显示状态组从一个 OpenGL 绘图描述表拷贝到另一个绘图描述表。函数的原型如下：

```
BOOL wglCopyContext( HGLRC hglrcSrc, HGLRC hlglrcDst, UINT mask );
```

它有 3 个参数：hglrcSrc、hglrcDst 和 mask。其中，hglrcSrc 参数用于指定要拷贝状态信息的源 OpenGL 绘图描述表，hglrcDst 参数指定要拷贝状态信息的目标 OpenGL 绘图描述表，mask 参数用于指定要拷贝到 hglrcDst 的 hglrcSrc 显示状态组，包括与 glPushAttrib 函数的参数相同符号名称的按值取逻辑“或”。可用 GL_ALL_ATTRIB_BITS 拷贝所有显示状态信息。

该函数有一个布尔量返回值，若调用函数成功，返回值是 TRUE；若调用函数失败，返回值是 FALSE。

注意： 用 wglCopycontext 可以使两个绘图描述表的显示状态同步，可以在相同线程的两个绘图描述表间仅拷贝显示状态。这些绘图描述表要来自相同的 OpenGL 程序，例如，可以在同一线程的两个有相同图像格式的绘图描述表间拷贝显示状态。

可以拷贝用 glPushAttrib 函数得到的相同状态信息，但不能拷贝如像素压缩/解压状态、显示模式状态、选择和反馈状态等信息。调用 wglCopyContext 函数时要确保 hglrcDst

目标绘图描述表不属于任何一个线程。

3.2.4 wglDeleteContext 函数

该函数的功能是删除一个指定的 OpenGL 绘图描述表。其原型为：

```
BOOL wglDeleteContext( HGLRC hglrc );
```

其中，hglrc 参数是要删除的 OpenGL 绘图描述表的句柄。

函数有一个布尔量返回值，若调用函数成功，返回值 TRUE；若调用函数失败，返回值为 FALSE。

注意： 删除另一线程的 OpenGL 当前绘图描述表是错误的。如果绘图描述表位于调用线程，则 wglDeleteContext 函数会在删除它之前，使当前的绘图描述表不是它。调用 wglMakeCurrent 函数时，wglDeleteContext 函数不删除与 OpenGL 绘图描述表相关的设备描述表。调用 wglDeleteContext 后，一定要调用 DeleteDC 删除相关的设备描述表。

3.2.5 wglDescribeLayerPlane 函数

该函数的功能是提供给定像素格式层平面的有关信息。函数的原型如下：

```
BOOL wglDescribeLayerPlane( HDC hdc ,  int iPixelFormat ,  int
iLayerPlane,  UINT nBytes,  LPLAYERPLANEDESCRIPTOR plpd );
```

它有 5 个参数：hdc，iPixelFormat，iLayerPlane，nBytes 和 plpd。其中，hdc 参数指定要描述的窗口层平面的设备描述表。iPixelFormat 参数指定要描述的层平面像素格式。iLayerPlane 参数指定层平面、层平面的号码，在像素格式描述符结构的 bReserved 成分中给出。nBytes 参数指定 plpd 指向的结构字节数。wglDescribeLayerPlane 函数在 LAYERPLANEDESCRIPTOR 结构中存储不大于 nBytes 的层平面数据，把这个结构的尺寸赋给 nBytes。plpd 参数是一个指向 LAYERPLANEDESCRIPTOR 结构的指针。wglDescribeLayerPlane 函数给结构的成分赋值，并把拷入结构的数据字节数存在 nSize 成分中。

该函数有一个布尔量返回值，若调用函数成功，返回值为 TRUE，并按 iLayerPlane 层的 iPixelFormat 像素格式给 plpd 指向的 LAYERPLANEDESCRIPTOR 结构成分赋值；若调用函数失败，返回值为 FALSE。

注意： 层平面的号码 iLayerPlane 决定了其顺序，号码大的平面总是覆盖在号码小的平面上。

3.2.6 wglGetcurrentContext 函数

该函数的功能是给正在调用线程的当前 OpenGL 绘图描述表提供一个句柄。其原型为：

```
HGLRC wglGetcurrentContext(void);
```

该函数不带任何参数。其返回值是一个结构指针，若正在调用的线程中有一个当前

OpenGL 绘图描述表，wglGetCurrentContext 函数返回一个绘图描述表的句柄，否则返回值是 NULL。

注意： 调用 wglGetcurrentContext 函数，可以使当前的 OpenGL 绘图描述表与设备描述表联系起来。可以用 wglGetCurrentDC 函数得到一个与当前 OpenGL 绘图描述表相关的设备描述表句柄。

3.2.7 wglGetCurrentDC 函数

该函数的功能是提供一个正在调用线程与当前 OpenGL 绘图描述表相关的设备描述表句柄。其原型如下所示：

```
HDC wglGetCurrentDC(void);
```

该函数不带任何参数。其返回值是一个设备描述表句柄，若正在调用线程有一个当前 OpenGL 绘图描述表，借助 wglGetcurrentContext 函数，该函数返回一个与绘图描述表相关的设备描述表句柄，否则，返回值是 NULL。

注意： 调用函数时，使设备描述表与 OpenGL 绘图描述表联系起来，可以用 wglGetCurrent-DC 函数得到一个正在调用线程 OpenGL 绘图描述表的句柄。

3.2.8 wglGetLayerPaletteEntries 函数

该函数的功能是从指定设备描述表的色彩指数层平面检索调色板的内容。其原型如下所示：

```
int wglGetLayerPaletteEntries( HDC hdc,  int iLayerPlane,  int iStart,
                               int cEntries,  CONST COLORREF *pcr );
```

该函数有 5 个参数：hdc，iLayerPlane，iStart，cEntries 和 pcr。其中，hdc 参数定义要描述层平面的窗设备描述表，iLayerPlane 参数指定层平面的号码，iStart 参数指定要检索的第一个调色板，cEntries 参数指定要检索的调色板条目数量，pcr 参数包含调色板 RBB 颜色值的 COLORREF 结构矩阵的指针，这个矩阵至少能容纳 cEntries 个结构。

该函数有一个整型返回值：若调用函数成功，返回值是指定窗口层平面的调色板数；若调用函数失败，或没选择像素格式，返回值是 0。

注意： 每个色彩指数的窗口层平面的调色板大小为 2^n，n 为每个层平面内的位平面数目，调色板的透明指数是不能改动的。可以用 wglRealizeLayerPalette 函数实现层调色板。初始时，层调色板只有白色。wglSetpaleteEntries 函数不能给主平面调色板赋值，只能用 GDI 函数改变主平面调色板。

3.2.9 wglGetProcAddress 函数

该函数的功能是返回当前 OpenGL 绘图描述表的 OpenGL 扩展函数的地址。其原型如下所示：

```
PROC wglGetProcAddress( LPCSTR lpszProc );
```

该函数有一个参数 lpszProc，用于指向扩展函数名的空结束字符串的指针，扩展函数名要与用 OpenGL 实现的相应扩展函数名相同。

该函数有一个函数指针型返回值，若调用函数成功，返回值为扩展函数的地址；若无当前绘图描述表或函数调用失败，则返回值为 NULL。

注意：OpenGL 库支持多种函数的实现，在一个绘图描述表支持的扩展函数不必在另外的绘图描述表中得到。对程序中一个给定的绘图描述表，可以用 wglGetProcAddress 函数返回的地址。由 lpszProc 指向扩展函数的拼写、大小写必须与 OpenGL 中支持并实现的函数相同。因为 OpenGL 不输出扩展函数，必须用 wglGetProcAddress 函数得到特定扩展函数的地址。

3.2.10 wglMakeCurrent 函数

该函数的功能是把一个指定的 OpenGL 绘图描述表变为正在调用线程的当前绘图描述表。其后这个线程的所有 OpenGL 函数调用，均输出到 hdc 提供的设备上，也可以用 wglMakeCurrent 函数把当前线程的绘图描述表变成非当前值。其原型如下所示：

```
BOOL wglMakeCurrent(HDC hdc, HGLRC hglrc);
```

该函数有 2 个参数：hdc 和 hglrc。其中，hdc 参数是设备描述表的句柄，hglrc 参数将函数设定为当前线程 OpenGL 绘图描述表的句柄。若 hglrc 为 NULL，则该函数把当前线程的绘图描述表设成非当前值，释放绘图描述表使用的设备描述表，并忽略 hdc。

该函数有一个布尔返回值，若调用函数成功，返回值 TRUE；若调用函数失败，返回值为 FALSE。

注意：hdc 参数一定要与 OpenGL 支持的绘图曲面有关，它不必是 hglrc 创建时传递给 wglCreateContext，但一定要在相同的设备上并且具有相同的像素格式。绘图描述表不支持 hdc 的 GDI 变换和裁剪，当前绘图描述表是 hdc 指向的设备描述表，到它不再是当前值为止。

在变成新的绘图描述表之前，OpenGL 要执行完当前调用线程的所有绘图描述表。绘图描述表在一个时刻只能是一个线程的当前值，不能是多个线程，而程序可以多线程绘图，每个线程有各自的绘图描述表和设备描述表。

若出现错误，wglMakeCurrent 函数在返回前，把线程的当前绘图描述表变成非当前值。

3.2.11 wglRealizeLayerPalette 函数

该函数的功能是把来自色彩指数层平面的调色板数映射到物理调色板或初始化 RGB 层平面。其原型如下所示：

```
BOOL wglRealizeLayerPalette(HDC hdc, int iLayerPlane, BOOL bRealize);
```

该函数带 3 个参数：hdc，iLayerPlane 和 bRealize。其中，hdc 参数用于在物理调色板实现层平面调色板的窗口设备描述表，iLayerPlane 参数是层平面编号，bRealize 参数标识是否在物理调色板上实现调色板。当 bRealize 为 TRUE，调色板条目映射到物理调色板；若 bRealize 为 FALSE，不再需要窗口层平面的调色板，可以释放给其他前台窗口使用。

该函数的返回值是一个布尔量，若调用函数成功，返回值 TRUE，即，使 bRealize 为 TRUE，但是没有物理调色板；若调用函数失败或没有选择像素格式，返回值为 FALSE。

注意： 对每个层平面、各窗口的层平面之间，物理调色板是一个共享的资源。一次只能实现一个调色板。若调用 wglRealizeLayerPalette 函数，前台窗口的层调色板总是最先实现。

实现窗口的层调色板后，其层调色板总是一对一映射到物理调色板。与 GDI 逻辑调色板不同，这个函数不会把其他窗口层调色板映射到当前物理调色板上。即使层平面的像素格式是 RGBA，只要窗口再变成前台窗口，调用 wglRealizeLayerPalette 可以再实现其层调色板。但是 wglRealizeLayerPalette 不实现主平面的调色板，只能用 GDI 调色板函数实现主调色板。

3.2.12　wglSetLayerPaletteEntries 函数

该函数的功能是为指定的设备描述表在给定色彩指数层平面上的调色板赋值。其原型如下所示：

```
int wglSetLayerPaletteEntries( HDC hdc,  int iLayerPlane,  int iStart,
int cEntries,  CONST COLORREF *pcr );
```

该函数有 5 个参数：hdc，iLayerPlane，iStart，cEntries 和 pcr。其中，hdc 是要赋值的层调色板的层窗口设备描述表，iLayerPlane 参数指定层平面的号码，iStart 参数指定要检索的第一个调色板，cEntries 参数指定要检索的调色板数量，pcr 参数包含调色板 RGB 颜色值的 COLORREF 结构矩阵的指针，这个矩阵至少能容纳 cEntries 个结构。

该函数的返回值是一个整型值，若调用函数成功，返回值是指定窗口层平面的调色板数；若调用函数失败，或没选择像素格式，返回值是 0。

注意： 每个色彩指数的窗口层平面的调色板大小为 2^n，n 为每个层平面内的位平面数目。调色板的透明指数是不能改动的，可以用 wglRealizeLayerPalette 函数实现层调色板。初始时，层调色板只有白色。wglSetLayerPaletteEntries 函数不能给主平面调色板赋值，只能用 GDI 函数改变主平面调色板。

3.2.13　wglShareLists 函数

该函数的功能是使多个 OpenGL 绘图描述表共享一个显示列表空间。其原型如下所示：

```
BOOL wglShareLists( HGLRC hglrc1,  HGLRC hglrc2 );
```

该函数有 2 个参数：hglrc1 和 hglrc2。其中，hglrc1 参数指定共享显示列表的 OpenGL 绘图描述表，hglrc2 参数指定与 hglrc1 共享显示列表的 OpenGL 绘图描述表。当调用 wglShareLists 时，hglrc2 参数不能包括任何已经存在的显示列表。

该函数返回值是一个布尔量，若调用函数成功，返回值 TRUE；若调用函数失败，返回值是 FALSE，显示列表不共享。

注意： 每个 OpenGL 绘图描述表都有各自的显示列表空间。wglShareLists 允许一个绘图描

述表与另一个绘图描述表共享显示列表空间，并且任意多个绘图描述表可以共享一个显示列表空间。一旦绘图描述表共享一个显示列表空间，在被删除前总可以使用这个空间。当删除最后一个共享显示列表空间的绘图描述表时，这个共享空间也就删除了。在其共享空间，显示列表的指数和定义均共享。

只能在相同线程的绘图描述表共享显示列表，只有用相同的 OpenGL 函数实现的绘图描述表才能共享显示列表，所有具有给定的像素格式的客户绘图描述表都能共享。所有共享的绘图描述表要用相同的像素格式，否则要看使用 OpenGL 实现的结果。

wglShareLists 是 OpenGL 1.01 或更新版中才有的函数，可以调用 glGetString 确定实现 OpenGL 的版本号。

3.2.14 wglSwapLayerBuffers 函数

该函数的功能是交换一个指定设备描述表的窗口覆盖层、底层及主平面的前后缓冲区。其原型如下所示：

```
BOOL wglSwapLayerBuffers( HDC hdc, UINT fuPlanes );
```

该函数带两个参数：hdc 和 fuPlanes。其中，hdc 参数指定要在物理调色板上实现层调色板的窗口设备描述表，fuPlanes 参数指定要交换前后缓冲区的覆盖层、底层及主平面。

PIXELFORMATDESCRIPTOR 结构的 bReserved 成分指定覆盖层、底层平面的号码，fuPlanes 是下列值的组合：

- WGL_SWAP_MAIN_PLANE　交换主平面的前后缓冲区。
- WGL_SWAP_OVERLAYi　交换第 i 个覆盖层平面的前后缓冲区。
- WGL_SWAP_UNDERLAYi　交换第 i 个底层平面的前后缓冲区。

该函数返回值是一个布尔量，若调用函数成功，返回值为 TRUE；若调用函数失败，返回值为 FALSE。

注意：*若一个层平面没有后缓冲区，wglSwapLayerBuffers 对其无影响。调用 wglSwapLayerBuffers 后，后缓冲区内容的状态赋给相应的层平面 LAYERPLANEDESCRIPTOR 结构或主平面的 PIXELFORMATDESCRIPTOR 结构。wglSwapLayerBuffers 同时交换指定层平面的前后缓冲区。*

一些设备不支持单独交换层平面，而是作为一组交换所有层平面。当设定了像素格式描述符的 PFD_SWAP_LAYER_BUFFERS 标识后，可以调用 wglSwapLayerBuffers 只交换层平面中的一个层平面。

3.2.15 wglUseFontBitmaps 函数

该函数的功能是在当前 OpenGL 的绘图描述表创建一系列位图显示列表，这些显示列表是基于设备描述表中当前选定的字体的字模。可以用这些位图在 OpenGL 图像中绘制出字符。wglUseFontBitmaps 创建 count 个显示列表，每个都对应 count 个字模中的一个字符，以 hdc 参数中选定字体的 first 字模开头。其原型如下所示：

```
BOOL wglUseFontBitmaps( HDC hdc, DWORD first, DWORD count, DWORD
```

```
    listBase );
```

该函数带 4 个参数：hdc，first，count 和 listBase。其中，hdc 参数指定当前 OpenGL 绘图描述表中用当前选定字体形成字模显示列表的设备描述表，first 参数指定形成字模位图显示列表的一系列字模中的第一个字模，count 参数指定形成字模位图显示列表的数量。该函数创建 count 个显示列表，每个列表对应序列中的一个字模。listBase 参数指定开始的显示列表。

函数的返回值是一个布尔量，若调用函数成功，返回值 TRUE；若调用函数失败，返回值为 FALSE。

注意：wglUseFontBitmaps 定义了当前 OpenGL 绘图描述表的 count 个显示列表。每个显示列表有一个标识号码，第一个显示列表的号码为 listBase，每个显示列表包含一个 glBitmap 调用，第 listBase+j 个位图的定义是由 hdc 指定的设备描述表当前选定的字体的第 first+j 个字模得到的。如果字模未定义，那么该函数定义了一个空的显示列表。

wglUseFontBitmaps 在屏幕的平面上创建位图文本，可以用于标注 OpenGL 的物体。虽然在现有的微软 Windows NT 和 Windows 95 的 OpenGL 版本中，不能对有双缓冲区像素格式的设备描述表使用 GDI 字体和文本函数，但是可以用 wglUseFontBitmaps 消除这个限制，并在双缓冲区画出文本。下面举例说明该函数的用法：

```
HDChdc;
HGLRC  hglrc;

// 创建绘制描述表
hglrc = wglCreateContext (hdc);

// 使之成为调用线程的当前绘制描述表
wglMakeCurrent (hdc,  hglrc);

// 调用 OpenGL API

// 使系统字体成为设备描述表的选中字体
SelectObject (hdc,  GetStockObject (SYSTEM_FONT));

// 创建位图显示列表
// 创建 0~25 字符图像
// 显示列表从 1000 开始编号，这是任意选取的数字
wglUseFontBitmaps (hdc,  0,  255,  1000);

// 显示一个字符串：
// 表明位图显示列表的起始值
glListBase (1000);
```

```
// 绘制字符串中的每一个字符
glCallLists (24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World");
```

3.2.16　wglUseFontOutlines 函数

该函数的功能是给当前绘图描述表创建一系列显示列表，每个显示列表对应一个当前设备描述表在浮点坐标中的轮廓字体，这些显示列表用于画三维 TrueType 字体的字符。其原型如下所示：

```
BOOL wglUseFontOutlines( HDC hdc, DWORD first, DWORD count, DWORD
listBase , FLOAT deviation , FLOAT extrusion , int format ,
LPGLYPHMETRICSFLOAT lpgmf );
```

该函数带 8 个参数：hdc，first，count，listBase，deviation，extrusion，format 和 lpgmf。其中，hdc 参数指定具有轮廓字体的设备描述表，hdc 的轮廓字体用于在当前绘图描述表创建显示列表。count 参数指定形成字模位图显示列表的数量，该函数创建 count 个显示列表，每个对应序列中的一个字模。first 参数指定形成字模位图显示列表的一系列字模中的第一个字模。listBase 参数指定开始的显示列表。deviation 参数定义与原轮廓的弦偏差。当偏差为 0 时，弦偏差相当于原字体一个单位，偏差值要大于等于 0。extrusion 参数定义字体向负 z 轴突出的大小。该值要大于等于 0，若其为 0，无突出。format 参数定义在显示列表中是应用 WGL_FONT_LINES 还是 WGL_FONT_POLYGONS 格式。若为 WGL_FONT_LINES 格式，wglUseFontOutlines 用线段创建字体；若是 WGL_FONT_POLYGONS 格式，则使用多边形。lpgmf 参数接受字模尺寸的 count 个 GLYPHMETRICSFLOAT 结构的矩阵指针。

函数的返回值是一个布尔量，若调用函数成功，返回值 TRUE；若调用函数失败，返回值为 FALSE，不产生显示列表。

注意： *wglUseFontOutlines 用显示列表定义当前绘图描述表轮廓字体的字模，只对 TrueType 字体适用，不支持笔划字体和光栅字体。*

每个显示列表由线段或多边形构成，都有唯一的以 listBase 开始的标识号码。当 format 为 WGL_FONT_LINES，wglUseFontOutlines 把字模轮廓的两次 B 样条曲线分成线段，使轮廓和插值中点的距离小于 deviation。当 format 为 WGL_FONT_POLYGONS，显示列表中没有法线，光照不正确，可以设置 glPolygonMode(GL_FRONT，GL_LINE)，使轮廓嵌入三角形、三角形扇形、三角形带或者四边形带，为轮廓构造表面。参数为 WGL_FONT_POLYGONS 创建的显示列表，调用 glFontBase(GL_CW) 或 glFrontBase(GL_CW)可能改变当前朝前面的值。这时，为使文本更好看，应该剔除朝后的面：

```
glCullFace(GL_BACK);
glEnable (GL_CULL_FACE);
```

下面说明 wglUseFontOutlines 函数的用法：

```
HDChdc;  //  TrueType 字体被选送
HGLRC  hglrc;
```

```
GLYPHMETRICSFLOAT agmf[256];

// 使 hglrc 成为调用线程的当前绘制描述表
wglMakeCurrent(hdc, hglrc);

// 创建以 0.1 为默认递加的 0~25518 的显示列表。显示列表编号从 1000 开始//
// (可以是任意值)
wglUseFontOutlines(hdc, 0, 255, 1000, 0.0f, 0.1f,
WGL_FONT_POLYGONS, &agmf);

// 设置绘制字符串的变形
glLoadIdentity();
glTranslate(0.0f, 0.0f, -5.0f)
glScalef(2.0f, 2.0f, 2.0f);

// 显示字符串
glListBase(1000); // 指出显示列表的起始值
// 绘制字符串中的每一个字符
glCallLists(24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World.");
```

3.3　与 OpenGL 相关的结构

Win32 环境下与 OpenGL 相关的结构有 4 个，以下分别对它们进行讲解。

3.3.1　GLYPHMETRICSFLOAT 结构

GLYPHMETRICSFLOAT 包含字符单元的位置、方向的信息。

```
typedef struct _GLYPHMETRICSFLOAT { // gmf
    FLOAT  gmfBlackBoxX;
    FLOAT  gmfBlackBoxY;
  POINTFLOAT gmfptGlyphOrigin;
  FLOAT  gmfCellIncX;
  FLOAT  gmfCellIncY;
} GLYPHMETRICSFLOAT;
```

结构的成分说明：

- gmfBlackBoxX：包括字模的最小矩形(字模的黑盒子)宽度。
- gmfBlackBoxY：包括字模的最小矩形(字模的黑盒子)高度。
- gmfptGlyphOrigin：包括字模的最小矩形左上角坐标(x，y)。

- gmfCellIncX：从当前字符单元到下一个字符单元的水平距离。
- gmfCellIncY：从当前字符单元到下一个字符单元的垂直距离。

注意：GLYPHMETRICSFLOAT 结构的值都用抽象单位。

3.3.2 LAYERPLANEDESCRIPTOR 结构

LAYERPLANEDESCRIPTOR 描述绘图曲面的像素格式。

```
typedef struct tagLAYERPLANEDESCRIPTOR { // pfd
    WORD  nSize;
    WORD  nVersion;
    DWORD dwFlags;
    BYTE  iPixelType;
    BYTE  cColorBits;
    BYTE  cRedBits;
    BYTE  cRedShift;
    BYTE  cGreenBits;
    BYTE  cGreenShift;
    BYTE  cBlueBits;
    BYTE  cBlueShift;
    BYTE  cAlphaBits;
    BYTE  cAlphaShift;
    BYTE  cAccumBits;
    BYTE  cAccumRedBits;
    BYTE  cAccumGreenBits;
    BYTE  cAccumBlueBits;
    BYTE  cAccumAlphaBits;
    BYTE  cDepthBits;
    BYTE  cStencilBits;
    BYTE  cAuxBuffers;
    BYTE  iLayerPlane
    BYTE  bReserved;
    COLORREF crTransparent;
} LAYERPLANEDESCRIPTOR;
```

结构的成分说明：

- nSize：数据结构的大小，把这个值赋为 sizeof(LAYERPLANEDESCRIPTOR)。
- nVersion：数据结构的版本号，这个参数设为 1。
- dwFlags：定义层平面属性的位标识，这些属性通常不相互排斥。如果不说明，可以赋成任何位标识组合，这些位标识常量是：

- ◆ LPD_SUPPORT_OPENGL：层平面支持绘图。
- ◆ LPD_SUPPORT_GDI：层平面支持 GDI 绘图，当前的 实现不支持这个标识。
- ◆ LPD_DOUBLEBUFFER：如果设置了该值，则即使主平面是单缓冲的，层平面也是双缓冲的；否则，即使主平面是双缓冲的，层平面也是单缓冲的。
- ◆ LPD_STEREO：如果设置了该值，则即使主平面是平面的层平面也是立体的；否则，即使主平面是立体的，层平面也是平面的。
- ◆ LPD_SWAP_EXCHANGE：在双缓冲的层平面，交换颜色缓冲同时也交换前后缓冲区内容，交换后，后缓仲区包含交换前的前缓冲区内容，这个标识是隐含的，不能由驱动程序提供。
- ◆ LPD_SWAP_COPY：在双缓冲的层平面，交换颜色缓冲同时把后缓冲区内容拷贝到前缓冲区，不影响后缓仲区的内容。这个标识是隐含的，不能由驱动程序提供。
- ◆ LPD_TRANSPARENT：本结构的 crTransparent 成分含有一个透明颜色或指数值，允许底层平层透过这层显示出来。除最底层平层，均有一个透明颜色或指数。
- ◆ LPD_SHARE_DEPTH：层平面与主平面共享深度缓冲区。
- ◆ LPD_SHARE_STENCIL：层平面与主平面共享板缓冲区。
- ◆ LPD_SHARE_ACCUM：层平面与主平面共享累加缓冲区。

- iPixelType 是两种像素数据类型之一：
  - ◆ LPD_TYPE_RGBA：RGBA 像素，每个像素有红、绿、蓝、alpha 等 4 个成分。
  - ◆ LPD_TYPE_COLORINDEX：色彩指数像素，每个像素有一个色彩指数值。
- cColorBits：每个颜色缓冲区的颜色位面数目。对于 RGBA 像素，不包括 alpha 位平面的颜色缓冲区的大小；对于色彩指数像素，是色彩指数缓冲区的大小。
- cRedBits：每个 RGBA 颜色缓冲区红色位平面数目。
- cGreenBits：每个 RGBA 颜色缓冲区绿色位平面数目。
- cBlueBits：每个 RGBA 颜色缓冲区蓝色位平面数目。
- cRedShift：每个 RGBA 颜色缓冲区的红色位平面的偏移量。
- cGreenShift：每个 RGBA 颜色缓冲区的绿色位平面的偏移量。
- cBlueShift：每个 RGBA 颜色缓冲区的蓝色位平面的偏移量。
- cAlphaBits：每个 RGBA 颜色缓冲区的 alpha 位平面数目。
- cAlphaShift：每个 RGBA 颜色缓冲区的 alpha 偏移量。
- cAccumBits：累加缓冲区的位平面总数。
- cAccumRedBits：累加缓冲区的红色平面总数。
- cAccumGreenBits：累加缓冲区的绿色平面总数。
- cAccumBlueBits：累加缓冲区的蓝色平面总数。
- cAccumAlphaBits：累加缓冲区的 alpha 平面总数。
- cDepthBits：深度(z 轴)缓冲区的深度值。

- cStencilBits：模板缓冲区的深度值。
- cAuxBuffers：辅助缓冲区的数目，不支持辅助缓冲区。
- iLayerType：层平面编号。
- bReserved：不使用，设为 0。
- crTransparent：设置 LPD_TRANSPARENT 标识后，透明的颜色或指数值，通常设为 0。

注意：有些层平面属性在 Microsoft GDI 软件实现的 OpenGL 中不支持，有些硬件制造商可能会增强 OpenGL 的部分内容，能够支持在基本功能中不支持的一些层平面属性。

3.3.3 PIXELFORMATDESCRIPTOR 结构

PIXELFORMATDESCRIPTOR 描述绘图曲面的像素格式。

```
typedef struct tagPIXELFORMATDESCRIPTOR { // pfd
    WORD  nSize;
    WORD  nVersion;
    DWORD dwFlags;
    BYTE  iPixelType;
    BYTE  cColorBits;
    BYTE  cRedBits;
    BYTE  cRedShift;
    BYTE  cGreenBits;
    BYTE  cGreenShift;
    BYTE  cBlueBits;
    BYTE  cBlueShift;
    BYTE  cAlphaBits;
    BYTE  cAlphaShift;
    BYTE  cAccumBits;
    BYTE  cAccumRedBits;
    BYTE  cAccumGreenBits;
    BYTE  cAccumBlueBits;
    BYTE  cAccumAlphaBits;
    BYTE  cDepthBits;
    BYTE  cStencilBits;
    BYTE  cAuxBuffers;
    BYTE  iLayerType;
    BYTE  bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
```

```
        DWORD dwDamageMask;
    } PIXELFORMATDESCRIPTOR;
```

- nSize：数据结构的大小，把这个值赋给 sizeof(PIXELFORMATDESCRIPTOR)。
- nVersion：数据结构的版本号，这个参数设为 1。
- dwFlags：定义像素格式属性的一系列位标识，这些属性通常不相互排斥。如果不说明，可设置成位标识的任何组合，这些位标识的说明是：
  - PFD_DRAW_TO_WINDOW ：缓冲区可以在窗口或设备表面上绘图。
  - PFD_DRAW_TO_BITMAP：缓冲区可以在内存位图绘图。
  - PFD_SUPPORT_GDI：缓冲区支持 GDI 绘图，当前的基本功能与 PFD_DOUBLE_BUFFER 相互排斥。
  - PFD_SUPPORT_OPENGL：缓冲区支持 OpenGL 绘图。
  - PFD_GENERIC_ACCELERATED：支持像素格式加速基本功能的设备驱动程序，若清除这个标识，设定 PFD_GENERIC_FORMAT，只能使用基本功能支持的像素格式。
  - PFD_GENERIC_FORMAT：支持像素格式同样是被当作基本功能的 GDI 软件实现。若清除这个标识，使用设备驱动程序或硬件支持像素格式。
  - PFD_NEED_PALETTE：缓冲区在调色板管理的设备上，用 RGBA 像素，这类像素需要逻辑调色板。取得最好效果，要根据 cRedBits，cRedShift，cGreenBits，cGreenShift 和 cBlueBits，cBlueShift 的值指定调色板的颜色，在调用之前的设备描述表上创建并实现调色板。
  - PFD_NEED_SYSTEM_PALETTE：对支持只有一个 256 色模式的硬件调色板的硬件，定义像素格式描述符。对于用硬件加速的这些系统，硬件调色板在 RGBA 模式下固定顺序，在色彩指数模式下要与逻辑调色板匹配设定这个标识，一定要用 SetSystemPaletteUse 完成逻辑调色板与系统调色板的一对一映射。若硬件支持多个硬件调色板并且设备驱动程序可以为其分配地址，通常要清除这个标识，基本像素格式不设置这个标识。
  - PFD_DOUBLE_BUFFER：使用双缓冲区，在当前基本功能下，与 PFD_SUPPURT_GDI 相互排斥。
  - PFD_STEREO：缓冲区是立体的，当前基本功能不支持这个标识。
  - PFD_SWAP_LAYERBUEFERS：标识是否可以只交换双缓冲覆盖或底层平面像素格式层平面中的一个层平面，否则所有层平面作为一组一起交换。设定这个标识，支持 wglSwapLayerBuffers 调用 ChoosePixelFormat 函数时，可以定义下列位标识：

    PFD_DEPTH_DONTCARE ：使像素格式可以有无深度缓冲区。

    PFD_DOUBLEBUFFER_DONTCARE ：使像素格式可以是单缓冲或双缓冲。

    PFD_STEREO_DONTCARE ：使像素格式为平面或立体的。

    使用 glAddSwapHintRectWIN 扩展函数，给 PIXELFORMATDESCRIPTOR 加入两个新的成分：

PFD_SWAP_COPY：在缓冲区交换后，定义双缓冲的主颜色平面上的后缓冲区内容。交换颜色缓冲区使后缓冲区拷贝到前缓冲区，后缓冲区的内容不受交换的影响。PFD_SWAP_COPY 只是隐含的，不能由驱动程序提供。

PFD_SWAP_EXCHANGE：在缓冲区交换后，定义双缓冲的主颜色平面上的后缓冲区内容。交换颜色缓冲区也交换前、后缓冲区的内容。交换后，后缓冲区的内容为交换前的前缓冲区。PFD_SWAP_EXCHANGE 只是隐含的，不能由驱动程序提供。

- cRedBits：每个 RGBA 颜色缓冲区的红色位平面数目。
- cGreenBits：每个 BGBA 颜色缓冲区的绿色位平面数目。
- cBlueBits：每个 RGBA 颜色缓冲区的蓝色位平面数目。
- cRedShift：每个 RGBA 颜色缓冲区的红色位平面的偏移量。
- cGreenShift：每个 RGBA 颜色缓冲区的绿色位平面的偏移量。
- cBlueShift：每个 RGBA 颜色缓冲区的蓝色位平面的偏移量。
- cAlphaBits：每个 RGBA 颜色缓冲区的 alpha 位平面数 9。
- cAlphaShift：每个 RGBA 颜色缓冲区的 alpha 偏移量。
- cAccumBits：累加缓冲区的位平面总数。
- cAccumRedBits：累加缓冲区的红色平面总数。
- cAccumGreenBits：累加缓冲区的绿色平面总数。
- cAccumBlueBits：累加缓冲区的蓝色平面总数。
- cAccumAlphaBits：累加缓冲区的 alpha 平面总数。
- cDepthBits：深度(z 轴)缓冲区的深度值。
- cStencilBits：模板缓冲区的深度值。
- cAuxBuffers：辅助缓神区的数目，不支持辅助缓冲区。
- iLayerType：不使用，早期的 OpenGL 实现使用这个参数。
- bReserved：定义覆盖及底层平面的编号，0~3 字节最多定义 15 个覆盖层平面，4~7 字节最多定义 15 个底层平面。
- dwLayerMask：不使用，早期的 OpenGL 实现使用这个参数。
- dwVisibleMask：若像素类型是 RGBA，定义底层平面的透明颜色或色彩指数。
- dwDamageMask：不使用，早期的 OpenGL 实现使用这个参数。

注意：有些层平面属性在 Microsoft GDI 软件实现的 OpenGL 中不支持，有些硬件制造商可能会增强 OpenGL 的部分内容，能够支持在基本功能中不支持的一些层平面属性。

3.3.4 POINTFLOAT 结构

包含一点的(x，y)坐标。

```
typedef struct _POINTFLOAT { // ptf
  FLOAT  x;
  FLOAT  y;
```

```
} POINTFLOAT;
```

结构的成分说明：

x，y：分别为一点的水平和垂直坐标。

第4章　OpenGL与Visual C++ 6.0事件编程

在Windows环境下编程必须掌握事件(或消息)的概念及其基本编程方法，在Visual C++ 6.0编程中尤其是这样。为了编写出与Windows结合紧密的图形软件，必须熟悉事件编程，即应该能够对各种事件进行正确的响应。本章将从一个简单而具体的实例出发，讲解如何将OpenGL编程与Windows环境下的事件响应紧密地结合起来，从而改善应用程序的用户界面。

4.1　简单编程思路

本节先简要介绍本章例子程序的功能，然后介绍包括用户界面设计在内的一般程序设计步骤。

4.1.1　程序功能简介

本章介绍如何将OpenGL编程与Visual C++ 6.0事件编程相结合，在Visual C++ 6.0事件处理程序中利用OpenGL进行图形绘制，从而真正地将OpenGL融入到程序中，使之与程序的其他部分有机地结合成为一个整体。

本章实例从MFC AppWizard出发，建立一个单文档应用程序，并建立2个视图：操作命令视图和图形绘制视图，它们是在一个窗口上的创建一个TSplitter类形成的。操作命令视图内有设置各种颜色的功能，包括，背景色、环境光、镜面反射光和漫反射光等颜色，还有设置各种OpenGL特性的功能，包括，是否使用反混淆、是否计算阴影、是否加光照，以及形体的绘制模式取顶点模式、线框模式还是实心填充模式等，还有激活和关闭动画的功能以及对绘制的形体进行缩放的功能等。在图形绘制视图内用辅助库绘制了一个球，它将随着命令操作区的各种操作发生变化，例如缩小、放大或者改变光照效果等。当鼠标在命令操作区内按下并拖动时，绘制的球体将随着鼠标的拖动而绕x，y轴旋转。

4.1.2　简要实现步骤

在Visual C++ 6.0中编程之前，需要先进行必要的界面设计，若用AppWizard产生程序框架，则可以利用其产生很大一部分程序代码，省去许多不必要的手工工作。除此之外，利用AppWizard也为界面设计提供了诸多方便。以下是本章实例程序中所用到的程序设

计的基本步骤。

(1) 利用 MFC AppWizard 建立一个单文档应用程序。

(2) 用资源编辑器制作一个命令操作对话框，作为命令操作视图的界面。

(3) 为命令操作对话框建立一个类。

(4) 在 CMainFrame 中创建一个 CSplitterWnd 类，使其分别包含操作命令视图和图形绘制视图。

(5) 为命令操作对话框内的每个操作构件编写事件处理程序，同时编写命令操作对话框自身的 Paint 函数，使之与每个操作构件的变化相对应。

(6) 为图形绘制的 CRenderView 视图编写绘制函数，使之能与命令操作视图的操作相应。

(7) 编写图形绘制视图的鼠标事件处理程序，使鼠标在图形绘制区内的拖动实现所绘制的形体的旋转(此处为用辅助库绘制的球)。

(8) 编写 OnTimer 事件处理程序，使之在操作命令视图中激活动画时实现反复旋转的功能。

4.2　实例过程

4.2.1　建立一个单文档

打开 File 菜单，选择 New 菜单项，在弹出的对话框中单击 projects，选择 MFC App Wizard，创建一个新应用程序，输入 OGLEvent 作为程序的名称。在第一步中选择单文档应用程序，单击 Finish 按钮。于是 Visual C++ 6.0 便产生了一个应用程序的框架，生成的源程序中包括：OGLEvent.CPP，MainFrm.CPP，OGLEventDoc.CPP，OGLEventView.CPP，OGLEventView.CPP，StdAFX.CPP，OGLEvent.RC 及各自的包含文件。在程序中未对 OGLEvent.CPP，OGLEventDoc.CPP，StdAFX.CPP 和 OGLEvent.RC 作任何改动。

4.2.2　用资源编辑器制作一个命令操作对话框

如图 4.1 所示是命令操作对话框的各个组成部分，可以很简单地对照着制作出来。

4.2.3　为命令操作对话框建立一个视图类

在资源编辑器内选中命令操作对话框，单击右键，弹出一个菜单，选择 ClassWizard 菜单项，在弹出的 Adding a Class 对话框中选择创建一个新类，接着弹出一个 New Class 对话框，将 Base Class 改为 CFormView，再输入建立视图类的名称：CFormCommandView，一个基于该对话框的视图类便建立起来了。还需要定义与对话框上每个构件相关的对象或数值变量。做到这一点并不难，在 MFC ClassWizard 对话框中，选择 Member Variables，

在 Control IDs 列表框中列出了所有的构件的 ID。选择其中任何一项单击，会弹出一个 Add Member Variable 对话框，Category 中表明为其定义一个构件对象还是简单的值变量，为每一个构件建立如下所示的构件对象或值变量：

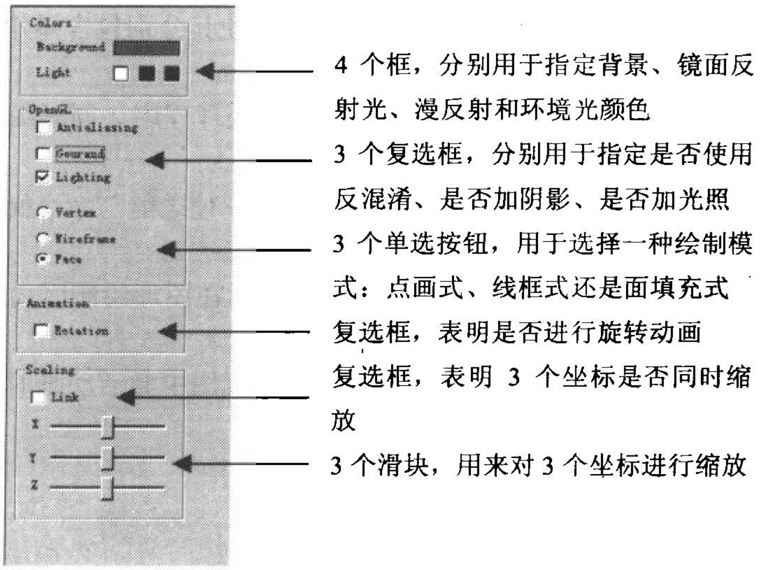

图4.1　操作命令对话框

```
CStatic  m_ControlColorLightSpecular;
CStatic  m_ControlColorLightDiffuse;
CSliderCtrl  m_SliderScaleZ;
CSliderCtrl  m_SliderScaleY;
CSliderCtrl  m_SliderScaleX;
CStatic  m_ControlColorLightAmbient;
CStatic  m_ControlBackColor;
BOOL  m_Lighting;
BOOL  m_Smooth;
BOOL  m_Antialias;
int  m_Model;
BOOL  m_VRotate;
BOOL  m_LinkScale;
```

4.2.4　在 CMainFrame 中创建一个 CSplitterWnd 类

先在 MainFrm.H 的公共属性中声明一个类：CSplitterWnd m_wndSplitter。CSplitterWnd 类提供了创建分区窗口的功能，即在一个窗口中建立多个分区视图，每个分区视图是一个从 Cview 继承下来的面板视图类，它可以有自己的子窗口 ID。完成上述声明后，接着可以编写 CMainFrame 类的 OnCreateClient 事件处理程序。先创建为一行两列的面板，再为

它们指定各自的视图类，这样两个视图的位置和大小就已经指定了，将把它们分别作为命令操作区和图形绘制区。

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
pContext)
{
// 创建一行两列的面板分区视图
    if (!m_wndSplitter.CreateStatic(this, 1, 2, WS_CHILD | WS_VISIBLE))
    {
    TRACE("Failed to CreateStaticSplitter\n");
    return FALSE;
    }
// 为第一个面板创建视图
    if (!m_wndSplitter.CreateView(0, 0,
    RUNTIME_CLASS(CFormCommandView), CSize(200, 160), pContext))
    {
      TRACE("Failed to create command view pane\n");
      return FALSE;
    }
    // 为第二个面板创建视图
    if (!m_wndSplitter.CreateView(0, 1,
      RUNTIME_CLASS(CRenderView), CSize(200, 200), pContext))
    {
      TRACE("Failed to create preview pane\n");
      return FALSE;
    }
    return TRUE;
}
```

4.2.5 编写命令操作对话框的事件处理程序

1. 鼠标事件处理程序

当鼠标在设置背景色、环境光、漫反射光和镜面反射光的 Frame 上单击时，用户希望程序能弹出一个颜色对话框供选择。但是 Frame 本身并没有接受鼠标消息的能力，鼠标在它上面的单击将直接传给其所在的窗体，因此需要编写鼠标在窗体上单击的事件处理程序。在处理程序中判断鼠标的位置，当它落在上述某一个 Frame 内，就弹出一个颜色对话框，并作相应的颜色设置，同时修改该 Frame 自身的颜色。以下就是窗口上鼠标单击事件的处理程序。

```
void CFormCommandView::OnLButtonUp(UINT nFlags, CPoint point)
```

```
{
  CRect rect;
  CToolApp *pApp = (CToolApp *)AfxGetApp();
  // 得到背景设置框的坐标
  m_ControlBackColor.GetWindowRect(&rect);
// 转换为屏幕坐标
  ScreenToClient(&rect);
// 判断鼠标单击点是否落在其中
  if(rect.PtInRect(point))
  {
// 弹出颜色设置对话框
    CColorDialog dlg(BackColor);
    if(dlg.DoModal()==IDOK)
    {
      BackColor = dlg.GetColor();
// 得到指向绘图区的指针
    CRenderView *pView = (CRenderView *)GetRenderView();
    PView->m_ClearColorRed=(float)GetRValue(BackColor)/ 255.0f;
    PView->m_ClearColorGreen=(float)GetGValue(BackColor)/255.0f;
    PView->m_ClearColorBlue=(float)GetBValue(BackColor) / 255.0f;
// 改变背景设置框的颜色
      this->InvalidateRect(&rect, FALSE);
// 改变绘图区的背景颜色
      pView->InvalidateRect(NULL, FALSE);
    }
  }
  // 环境光设置
  m_ControlColorLightAmbient.GetWindowRect(&rect);
  ScreenToClient(&rect);
  if(rect.PtInRect(point))
  {
    CColorDialog dlg(AmbientColor);
    if(dlg.DoModal()==IDOK)
    {
     AmbientColor = dlg.GetColor();
     CRenderView *pView = (CRenderView *)GetRenderView();
     // 用得到的颜色值修改 light0 的各颜色分量
     float r = (float)GetRValue(AmbientColor) / 255.0f;
     float g = (float)GetGValue(AmbientColor) / 255.0f;
```

```
        float b = (float)GetBValue(AmbientColor) / 255.0f;
        float  ambientProperties[]  = {r, g, b, 1.0f};
        glLightfv( GL_LIGHT0,  GL_AMBIENT,  ambientProperties);
        //  改变环境光设置框的颜色并按新的环境光重新绘制图形
        this->InvalidateRect(&rect, FALSE);
        pView->InvalidateRect(NULL, FALSE);
    }
}

//  镜面反射光设置
m_ControlColorLightSpecular.GetWindowRect(&rect);
ScreenToClient(&rect);
if(rect.PtInRect(point))
{
    CColorDialog dlg(SpecularColor);
    if(dlg.DoModal()==IDOK)
    {
        SpecularColor = dlg.GetColor();
        CRenderView *pView = (CRenderView *)GetRenderView();
        float r = (float)GetRValue(SpecularColor) / 255.0f;
        float g = (float)GetGValue(SpecularColor) / 255.0f;
        float b = (float)GetBValue(SpecularColor) / 255.0f;
        float  specularProperties[]  = {r, g, b, 1.0f};
        glLightfv( GL_LIGHT0,  GL_DIFFUSE,  specularProperties);
        // 改变镜面反射光设置框的颜色并按新的镜面反射光重新绘制图形
        this->InvalidateRect(&rect, FALSE);
        pView->InvalidateRect(NULL, FALSE);
    }
}

//  漫反射光颜色设置
m_ControlColorLightDiffuse.GetWindowRect(&rect);
ScreenToClient(&rect);
if(rect.PtInRect(point))
{
    CColorDialog dlg(DiffuseColor);
    if(dlg.DoModal()==IDOK)
    {
        DiffuseColor = dlg.GetColor();
```

```
            CRenderView *pView = (CRenderView *)GetRenderView();
            float r = (float)GetRValue(DiffuseColor) / 255.0f;
            float g = (float)GetGValue(DiffuseColor) / 255.0f;
            float b = (float)GetBValue(DiffuseColor) / 255.0f;
            float diffuseProperties[] = {r, g, b, 1.0f};
            glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
            // 改变漫反射光设置框的颜色并按新的漫反射光重新绘制图形
            this->InvalidateRect(&rect, FALSE);
            pView->InvalidateRect(NULL, FALSE);
        }
    }
    CFormView::OnLButtonUp(nFlags, point);
}
```

2. 反走样、加阴影和加光照3个OnCheck 事件的处理程序

这 3 个事件处理程序较为简单，其中 3 个变量 m_Lighting，m_Smooth 和 m_Antialias 分别用于记载 3 个 CheckBox 的状态。当 m_Lighting 状态为 On 时，激活光照；当 m_Smooth 状态为 On 时，使用 GL_SMOOTH 模式绘制图形，否则使用 GL_FLAT 模式；当 m_Antialias 状态为 On 时，采用反走样方式绘制，同时激活透明度作用并将线宽设为 1.5。设置完上述 OpenGL 状态后，再调用图形绘制区视图的 InvalidateRect 函数让绘图区重新绘制就可以了。

```
// 打开/关闭光照
void CFormCommandView::OnCheckLighting()
{
  m_Lighting = !m_Lighting;
  if(m_Lighting)
    glEnable(GL_LIGHTING);
  else
    glDisable(GL_LIGHTING);

  this->GetRenderView()->InvalidateRect(NULL, FALSE);
}

// 打开/关闭阴影
void CFormCommandView::OnCheckSmooth()
{
  m_Smooth = !m_Smooth;
  if(m_Smooth)
    glShadeModel(GL_SMOOTH);
```

```
    else
      glShadeModel(GL_FLAT);
    this->GetRenderView()->InvalidateRect(NULL, FALSE);

}

// 是否采用反走样方式绘制
void CFormCommandView::OnCheckAntialias()
{
  m_Antialias = !m_Antialias;
  if(m_Antialias)
  {
    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
    glLineWidth(1.5f);
  }
  else
  {
    glDisable(GL_LINE_SMOOTH);
    glDisable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
    glLineWidth(1.0f);
  }
GetRenderView()->InvalidateRect(NULL, FALSE);
}
```

3. 点画式、线框式和面填充3种绘图模式的选择

OpenGL 有 3 种绘制多边形的模式，即点画式、线框式和面填充模式，3 种模式分别对应着一个 RadioBox，将这 3 个 RadioBox 设置为属于同一组，因此在任何时候只能选择其中一种模式。

```
// 点画式
void CFormCommandView::OnRadioModel0()
{
  glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
  this->GetRenderView()->InvalidateRect(NULL, FALSE);
}
```

```
// 线框式
void CFormCommandView::OnRadioModel1()
{
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
  this->GetRenderView()->InvalidateRect(NULL, FALSE);
}

// 面填充式
void CFormCommandView::OnRadioModel2()
{
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  this->GetRenderView()->InvalidateRect(NULL, FALSE);
}
```

4. 用3个Slider控制形体缩放比例

首先将 3 个 Slider 的活动范围设置为 1~100，并使活动杆位置位于正中间，即 50。

```
void CFormCommandView::OnInitialUpdate()
{
  CFormView::OnInitialUpdate();
  // Slider
  TRACE("Sliders : updating...\n");
  m_SliderScaleX.SetRange(1, 100, TRUE);
  m_SliderScaleY.SetRange(1, 100, TRUE);
  m_SliderScaleZ.SetRange(1, 100, TRUE);
  m_SliderScaleX.SetPos(50);
  m_SliderScaleY.SetPos(50);
  m_SliderScaleZ.SetPos(50);
}
```

当活动杆在滑动条上移动时，重新计算缩放比例，若 3 个坐标同时变化的 CheckBox 状态为 On，则将 y，z 坐标对应的滑动条上的活动杆位置按 x 坐标对应的滑动条进行设置。

```
void CFormCommandView::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
  UpdateScale();
  GetRenderView()->InvalidateRect(NULL, FALSE);
  CFormView::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

```
BOOL CFormCommandView::UpdateScale()
{
  CRenderView *pView = (CRenderView *)GetRenderView();
//  重新计算缩放比例
  pView->m_xScaling = (float)m_SliderScaleX.GetPos()/50.0f;
  pView->m_yScaling = (float)m_SliderScaleY.GetPos()/50.0f;
  pView->m_zScaling = (float)m_SliderScaleZ.GetPos()/50.0f;
//  若 3 个坐标连动
  if(m_LinkScale)
  {
//  将 y, z 坐标按 x 坐标进行设置
    m_SliderScaleY.SetPos(m_SliderScaleX.GetPos());
    m_SliderScaleZ.SetPos(m_SliderScaleX.GetPos());
    pView->m_yScaling = pView->m_zScaling = pView->m_xScaling;
  }
  return TRUE;
}
```

当设置 3 个坐标连动状态时，需将 y、z 坐标的缩放比例按 x 坐标的缩放比例重新设置，再产生图形绘制。

```
void CFormCommandView::OnCheckLinkScale()
{
  m_LinkScale = !m_LinkScale;
  if(m_LinkScale)
  {
    CRenderView *pView = (CRenderView *)GetRenderView();
    m_SliderScaleY.SetPos(m_SliderScaleX.GetPos());
    m_SliderScaleZ.SetPos(m_SliderScaleX.GetPos());
    pView->m_yScaling = pView->m_zScaling = pView->m_xScaling;
  }
  m_SliderScaleY.EnableWindow(!m_LinkScale);
  m_SliderScaleZ.EnableWindow(!m_LinkScale);
  GetRenderView()->InvalidateRect(NULL, FALSE);
}
```

5. 视图的OnPaint事件处理程序

该事件处理程序中主要是对设置背景色、环境光、漫反射光和镜面反射光的四个 Frame 按其选择的颜色重新填充，使其能表示当前的各项设置。

```
void CFormCommandView::OnPaint()
{
  // 用于显示的设备描述表
  CPaintDC dc(this);
  CRect rect;
  // 按所选择的颜色重新绘制背景色设置框
  m_ControlBackColor.GetWindowRect(&rect);
  ScreenToClient(&rect);
  CBrush BrushBack(BackColor);
  dc.FillRect(&rect, &BrushBack);
  // 按所选择的颜色重新绘制环境光设置框
  m_ControlColorLightAmbient.GetWindowRect(&rect);
  ScreenToClient(&rect);
  CBrush BrushLightAmbient(AmbientColor);
  dc.FillRect(&rect, &BrushLightAmbient);
  // 按所选择的颜色重新绘制漫反射光设置框
  m_ControlColorLightDiffuse.GetWindowRect(&rect);
  ScreenToClient(&rect);
  CBrush BrushLightDiffuse(DiffuseColor);
  dc.FillRect(&rect, &BrushLightDiffuse);
  // 按所选择的颜色重新绘制镜面反射光设置框
  m_ControlColorLightSpecular.GetWindowRect(&rect);
  ScreenToClient(&rect);
  CBrush BrushLightSpecular(SpecularColor);
  dc.FillRect(&rect, &BrushLightSpecular);
}
```

6. 打开形体旋转动画

打开形体旋转动画就是将图形绘制区的定时器打开，关闭形体旋转动画也就是将该定时器关闭。程序中首先得到图形绘制区视图的指针，再用它去打开定时器。得到图形绘制区视图指针的函数前面也曾多次用到，现举例如下：

```
// 打开形体旋转动画
void CFormCommandView::OnCheckVrotation()
{
  m_VRotate = !m_VRotate;
```

```
    CRenderView *pView = (CRenderView *)GetRenderView();
    if(m_VRotate)
      pView->SetTimer(1, 10, NULL);
    else
      pView->KillTimer(1);
}
//  得到图形绘制区视图指针的函数
CView *CFormCommandView::GetRenderView()
{
  CToolApp *pApp = (CToolApp *)AfxGetApp();
  CMainFrame *pFrame = (CMainFrame *)pApp->m_pMainWnd;
  CView *pView = (CView *)pFrame->m_wndSplitter.GetPane(0, 1);
  return pView;
}
```

7. 为图形绘制视图编写绘制函数

● 构造函数与析构函数

构造函数中进行了一些初始化的工作，如下所示。由于对析构函数未作任何改动，直接继承下来，此处不再列出。

```
CRenderView::CRenderView()
{
  //  OpenGL 绘制描述表及像素格式
  m_hGLContext = NULL;
  m_GLPixelIndex = 0;
  //  鼠标状态
  m_LeftButtonDown = FALSE;
  m_RightButtonDown = FALSE;
  m_CursorRotation = AfxGetApp()->LoadCursor (IDC _ CURSOR _
ROTATION);
  //  OpenGL 初始化
  CToolApp *pApp = (CToolApp *)AfxGetApp();
  InitGeometry();
}
```

为了程序清晰，将程序中与形体几何相关的初始化工作独立写成一个函数，这里包括 x，y 方向的旋转角度、x，y，z 方向的平移，以及 3 个坐标方向的缩放比例等。建议用户编程时也应养成这样的良好习惯，即尽量将程序组织得清晰易读，把相关性强的程序组织在一起。

```
void CRenderView::InitGeometry(void)
{
```

```
    m_xRotation = 0.0f;
    m_yRotation = 0.0f;
    m_xTranslation = 0.0f;
    m_yTranslation = 0.0f;
    m_zTranslation = -5.0f;
    m_xScaling = 1.0f;
    m_yScaling = 1.0f;
    m_zScaling = 1.0f;
}
```

- OnCreate 事件处理程序

该程序中将设置像素格式并创建 OpenGL 绘制描述表等，而只有创建了 OpenGL 绘制描述表和设置了像素格式后，OpenGL 的绘制命令才会有效，因此该事件处理程序是 Windows 环境下编写基于窗口系统和事件编程的 OpenGL 程序所必备的。该程序中还对 OpenGL 的一些默认属性进行了设置，如绘图模式、光照及材质属性等。

```
int CRenderView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CView::OnCreate(lpCreateStruct) == -1)
    return -1;
  // 得到视图的窗口句柄和设备描述表
  HWND hWnd = GetSafeHwnd();
  HDC hDC = ::GetDC(hWnd);
// 设置像素格式
  if(SetWindowPixelFormat(hDC)==FALSE)
    return 0;
  // 创建 OpenGL 绘制描述表
  if(CreateViewGLContext(hDC)==FALSE)
    return 0;
  // 设置默认绘图模式
  glPolygonMode(GL_FRONT, GL_LINE);
  glPolygonMode(GL_BACK, GL_LINE);
  glShadeModel(GL_FLAT);
  glEnable(GL_NORMALIZE);
  // 光照及材质属性
  GLfloat ambientProperties[]  = {0.7f, 0.7f, 0.7f, 1.0f};
  GLfloat diffuseProperties[]  = {0.8f, 0.8f, 0.8f, 1.0f};
  GLfloat specularProperties[] = {0.0f, 0.8f, 0.2f, 1.0f};
  glClearDepth( 1.0 );
  glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
  glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
```

```
    glLightfv( GL_LIGHT0, GL_SPECULAR, specularProperties);
    glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0f);
    // 默认情况下打开光照
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    return 0;
}
```

- 设置像素格式并创建 OpenGL 绘制描述表

设置像素格式函数的具体用法在第 3.1 节中已经详细地作了介绍，这里不再赘述。创建 OpenGL 绘制描述表的工作由 wglCreateContext 函数和 wglMakeCurrent 函数完成，前者创建一个 OpenGL 绘制描述表，后者将其设置为当前绘制描述表。这样，在此之后的所有 OpenGL 绘图命令所产生的绘制内容将显示到该视图中。

```
BOOL CRenderView::SetWindowPixelFormat(HDC hDC)
{
    PIXELFORMATDESCRIPTOR pixelDesc;
    // 像素格式属性设置
    pixelDesc.nSize = sizeof(PIXELFORMATDESCRIPTOR);
    pixelDesc.nVersion = 1;
    pixelDesc.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
      PFD_DOUBLEBUFFER | PFD_STEREO_DONTCARE;
    pixelDesc.iPixelType = PFD_TYPE_RGBA;
    pixelDesc.cColorBits = 32;
    pixelDesc.cRedBits = 8;
    pixelDesc.cRedShift = 16;
    pixelDesc.cGreenBits = 8;
    pixelDesc.cGreenShift = 8;
    pixelDesc.cBlueBits = 8;
    pixelDesc.cBlueShift = 0;
    pixelDesc.cAlphaBits = 0;
    pixelDesc.cAlphaShift = 0;
    pixelDesc.cAccumBits = 64;
    pixelDesc.cAccumRedBits = 16;
    pixelDesc.cAccumGreenBits = 16;
    pixelDesc.cAccumBlueBits = 16;
    pixelDesc.cAccumAlphaBits = 0;
    pixelDesc.cDepthBits = 32;
    pixelDesc.cStencilBits = 8;
    pixelDesc.cAuxBuffers = 0;
    pixelDesc.iLayerType = PFD_MAIN_PLANE;
```

```
    pixelDesc.bReserved = 0;
    pixelDesc.dwLayerMask = 0;
    pixelDesc.dwVisibleMask = 0;
    pixelDesc.dwDamageMask = 0;
    // 按上面设置的属性选择像素格式
    m_GLPixelIndex = ChoosePixelFormat(hDC, &pixelDesc);
    if(m_GLPixelIndex == 0) // 默认选择
    {
      m_GLPixelIndex = 1;
      if(DescribePixelFormat(hDC, m_GLPixelIndex,
        sizeof(PIXELFORMATDESCRIPTOR), &pixelDesc)==0)
        return FALSE;
    }
    // 设置像素格式
    if(!SetPixelFormat(hDC, m_GLPixelIndex, &pixelDesc))
      return FALSE;
    return TRUE;
}

BOOL CRenderView::CreateViewGLContext(HDC hDC)
{
// 创建绘制描述表
  m_hGLContext = wglCreateContext(hDC);
  if(m_hGLContext==NULL)
    return FALSE;
  // 使之成为当前绘制描述表
  if(wglMakeCurrent(hDC, m_hGLContext)==FALSE)
    return FALSE;
  return TRUE;
}
```

- OnPaint 和 OnSize 事件处理程序

OnPaint 事件处理程序是所有 OpenGL 绘制程序的真正源头，所有绘制命令都直接或间接地由该函数调用。作为一个示例，这里借助辅助库绘制了一个实心球体。

```
void CRenderView::OnPaint()
{
  // 设备描述表
  CPaintDC dc(this);
  // 在单文档应用程序中需用以下语句得到 OpenGL 绘制描述表
  HWND hWnd = GetSafeHwnd();
```

```
    HDC hDC = ::GetDC(hWnd);
    wglMakeCurrent(hDC, m_hGLContext);
    //  清除颜色缓存和深度缓存
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //  用背景设置框中设置的颜色清除窗口
glClearColor(m_ClearColorRed , m_ClearColorGreen , m_ClearColor-
Blue, 1.0f);
    glPushMatrix();
    //  平移、 旋转和缩放，这里的各项参数可能已被操作命令区中的操作修改
    glTranslated(m_xTranslation, m_yTranslation, m_zTranslation);
    glRotatef(m_xRotation, 1.0f, 0.0f, 0.0f);
    glRotatef(m_yRotation, 0.0f, 0.0f, 1.0f);
    glScalef(m_xScaling, m_yScaling, m_zScaling);
    //  绘制一个实心球体(当多边形绘制设置为线框模式时亦将成为网状)
    auxSolidSphere(1.0);
    glPopMatrix();
    //  为避免动画时闪动，采用双缓存
    SwapBuffers(hDC);
}
```

OnSize 事件在窗口改变大小时发生。当窗口大小改变时，为了避免比例失调，需要对视口和显示比例进行一些调整，用 glViewport 函数指定视口矩形，用 gluPerspective 函数指定仰角、宽高比和深度裁剪值，gluPerspective 函数相当于指定了一个视棱台。

```
void CRenderView::OnSize(UINT nType, int cx, int cy);
{
    CView::OnSize(nType, cx, cy);

    // Set OpenGL perspective, viewport and mode
    CSize size(cx, cy);
    double aspect;
aspect=(cy=0)?(double)size.cx : (double)size.cx/(double)size.cy;

    glViewport(0, 0, size.cx, size.cy);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45, aspect, 1, 15.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glDrawBuffer(GL_BACK);
    glEnable(GL_DEPTH_TEST);
```

```
}
```

8. 图形绘制视的鼠标事件处理程序

设计鼠标的功能为：当鼠标在图形绘制视图中按下并拖动时，绘制的形体(此处为一个实心的球体)随着鼠标的移动而绕 x，y 轴旋转。旋转的角度跟鼠标在 x ，y 方向上的位移有关，鼠标在 x 方向上的位移决定绕 x 轴的旋转角度，鼠标在 y 方向上的位移决定绕 y 轴的旋转角度。因此当鼠标左键按下时，标记为按下状态并记下位置，以便鼠标移动时计算位移。鼠标释放时需将鼠标按下状态重新置为 FALSE。

```
void CRenderView::OnLButtonDown(UINT nFlags, CPoint point)
{
  m_LeftButtonDown = TRUE;
  m_LeftDownPos = point;
  CView::OnLButtonDown(nFlags, point);
}

void CRenderView::OnLButtonUp(UINT nFlags, CPoint point)
{
  m_LeftButtonDown = FALSE;
  CView::OnLButtonUp(nFlags, point);
}
```

在鼠标移动时，只需简单地计算当前鼠标位置与保存的位置之间的差，即为鼠标的位移，然后再将位移值除以 3.0 作为旋转角度。这时再保存鼠标的当前位置，以便鼠标移到一个新位置时重新计算位移。最后，调用 InvalidateRect 函数，相当于让系统去调用图形绘制区视图的 OnPaint 事件处理程序，这样，我们对上述参数的修改就会立即对绘制产生影响。

```
void CRenderView::OnMouseMove(UINT nFlags, CPoint point)
{
  if(m_LeftButtonDown)
  {
    m_xRotation -= (float)(m_LeftDownPos.x - point.x)/3.0f;
    m_yRotation -= (float)(m_LeftDownPos.y - point.y)/3.0f;
    m_LeftDownPos = point;
    InvalidateRect(NULL, FALSE);
  }
  CView::OnMouseMove(nFlags, point);
}
```

9. OnTimer 事件处理程序

OnTimer 事件每隔固定的时间会产生一次，因此经常被用来实现各种动画效果，这里先用它来修改绕 y 轴的旋转角度，再调用 InvalidateRect 函数使旋转生效。这样当操作命令视图中允许动画的 CheckBox 状态为 On，即定时器被打开时，图形绘制区的球体每隔 10ms 就会绕 y 轴旋转 5°，从而产生了不断地绕 y 轴旋转的动画效果。

```
void CRenderView::OnTimer(UINT nIDEvent)
{
  switch(nIDEvent)
    {
    case 0:
      break;
    // Rotation
    case 1:
      m_yRotation += 5.0f;
      InvalidateRect(NULL, FALSE);
      break;
    default:
      {}
    }
}
```

10. 程序运行结果

程序运行的结果如图 4.2 所示，可以看到，这是一个非常美观的界面，因此后面的几个实例我们仍然使用该界面，或稍加修改。至此，已将 OpenGL 的图形绘制功能与通用界面编程融为一体。

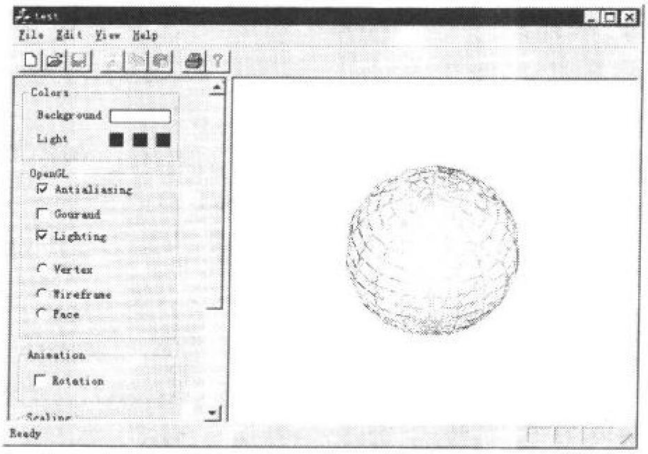

图4.2　程序运行结果示意图

第 5 章　基本几何图元的定义

基本几何图元的构造是 OpenGL 一切后续处理的基础，因此对这部分内容的掌握是学习 OpenGL 的基石，当然，这部分相对于本书其他部分的内容也是最简单的。通过本章的学习，将轻松地掌握以下内容：

- 如何开始和结束 OpenGL 绘图。
- 点、线与多边形的简单定义。
- 特殊多边形的绘制方法。
- 点、线与多边形的属性设置。
- 图形学的一些基本运算，例如法向量的计算方法等。
- 利用简单几何图元构造较复杂形体的方法。

5.1　绘制初始化命令

由于 OpenGL 是在窗口系统下运行的一种三维图形 API，因此在绘制之前必须对窗口进行一些初始化操作，例如清除窗口及设置窗口背景色等。而在 OpenGL 的绘制命令之后，还需要加上结束绘图命令，才能使绘制的图形真正地显示到窗口中。

OpenGL 的清除命令有以下几个：

- void glClearColor(GLfloat red，GLfloat green， GLfloat blue， GLfloat alpha)；

该命令用于指定颜色缓存的清除值，4 个参数分别指定红、绿、蓝分量及透明度值。

- void glClearIndex(GLfloat c)；

该命令用于指定颜色索引缓存的清除值，即指定调用 glClear 时的颜色缓存的清除值。其默认值为 0.0。

- void glClearDepth(GLfloat depth)；

该命令用于指定深度缓存的清除值，depth 的取值范围是[0.0， 1.0]，默认值为 0.0。

- void glClearAccum(GLfloat red， GLfloat green， GLfloat blue， Glfloat alpha)；

该命令用于指定累积缓存的清除值，4 个参数的取值范围是[－1.0， 1.0]，默认值均为 0.0。

- void glClearStencil(GLint s)；

该命令用于指定模板缓存的清除索引值，默认值亦为 0。

- void glClear(GLbitfield mask)；

该命令用于清除视区里的一种或多种缓存，缓存的清除值由上述的几个函数指定。若在此之前未调用上述相应的设置清除值的函数，则清除时使用相应的默认值。参数 mask 决定所要清除的缓存。mask 取值含义如下：

- GL_COLOR_BUFFER_BIT　　颜色缓存
- GL_DEPTH_BUFFER_BIT　　深度缓存
- GL_ACCUM_BUFFER_BIT　　累积缓存
- GL_STENCIL_BUFFER_BIT　　模板缓存

该命令可以一次清除多个缓存，只需在调用该函数时 mask 的取值为上述某些指定缓存的“或”即可。例如，若想要同时清除颜色缓存和深度缓存，可以这样调用该函数：

```
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
```

关于以上各种缓存的使用，将在后面的实例中详细地讨论。

在用 OpenGL 绘制对象之前，通常要先设定将要绘制的颜色或颜色方式，从而达到较高的绘制性能。设置颜色可以调用函数：glColor3f()。

这个命令利用 3 个参数，依次为红、绿、蓝分量，全部分量为在[0.0，1.0]之间的浮点数。1.0 表示最亮的颜色值，0.0 表示最暗的颜色值。最终设置的颜色相当于这 3 个颜色分量叠加的结果。例如，可以用下面几组调用设置几种常见的颜色：

- glColor(1.0，0.0，0.0);　　红色
- glColor(0.0，1.0，0.0);　　绿色
- glColor(0.0，0.0，1.0);　　蓝色
- glColor(0.0，0.0，0.0);　　黑色
- glColor(0.0，1.0，1.0);　　青色
- glColor(1.0，1.0，0.0);　　黄色
- glColor(1.0，0.0，1.0);　　品红
- glColor(1.0，1.0，1.0);　　白色

OpenGL 有两条结束绘图命令：void glFlush(void)和 void glFinish(void)，两者的区别是，前者将在有限的时间内执行前面发出的 GL 命令，而后者则等到完成前面发出的全部 GL 命令后再返回。这里所说的完成全部命令包括完成所有 GL 状态、联接状态的修改，以及对于帧缓存的修改等。在显示连续多帧画面时，前者的某些帧可能会只绘出部分帧来，而后者由于每一帧都绘完才返回，因此帧速率一般比利用前者绘制时要低。

5.2　基本图元的绘制

5.2.1　点、线与多边形的简单定义

在 OpenGL 中，所有几何物体最终都是由一定顺序的顶点集合来描述的。可以用 glVertex{234}{sifd}[v](TYPE coords)函数，来定义一个二维或三维或者一个用齐次坐标定义的顶点。例如，可以定义如下一组顶点：

```
glFloat vp[2]= (20.4, 17.8, 21.6);
    glBegin(GL_POINT);
glVertex2f( 2.0f, 3.0f );
```

```
glVertex3d( 2.0,   3.1215927,  1.000023 );
glVertex4f( 1.0f,  2.0f,  3.0f,  4.0f );
glVertex3fv( vp );
   glEnd();
```

第 1 条命令表示定义一个空间顶点(2.0f, 3.0f, 0.0f)，第 2 条命令表示用双精度浮点值定义一个三维空间顶点。第 3 条命令表示用齐次坐标定义一个顶点，顶点的真实坐标为：(0.25, 0.5, 0.75)。第 4 条命令演示了如何用一个数组来定义一个顶点。

连续地调用顶点定义函数可以生成一系列顶点、线或多边形等，在 OpenGL 中所有被定义的顶点必须放在 glBegin 和 glEnd 之间，才能正确地表达一个几何图元和物体，否则 OpenGL 不完成任何操作。既然一系列顶点可以生成不同类型的几何图元，为了区分，需要设定调用 glBegin 时的参数来指定用这些顶点将生成何种几何图元。glBegin 共有 10 种不同的参数。glBegin 的定义形式为 glBegin(GLenum mode)，如表 5.1 所示为这 10 种参数的名称及其含义。

表5.1 glBegin()参数的名称及其含义

名 称	含 义
GL_POINT	离散的顶点集合
GL_LINES	多组线段
GL_POLYGON	单个简单填充多边形
GL_TRIANGLES	多组独立填充三角形
GL_QUADS	多组独立填充四边形
GL_LINE_STRIP	不闭合折线
GL_LINE_LOOP	闭合折线
GL_TRIANGLE_STRIP	连续填充三角形序列
GL_TRIANGLE_FAN	连续填充三角形扇
GL_QUAD_STRIP	连续填充四边形序列

上述 10 种图元中大多数图元的含义比较明确，不再赘述，有关三角形和四边形的几组图元含义容易混淆，下面分别对它们加以比较。

连续填充三角形是一系列以公共边相连接的三角形。它由 glBegin()和 glEnd()之间的一系列顶点加以定义，最先给出的 3 个顶点定义了第一个三角形(这 3 个顶点是有序的，不同的顺序将影响生成的最终结果)，随后每给出一个顶点相当于增加了一个三角形，新三角形由新顶点与上一个三角形中 2、3 两个顶点组成，依此类推，便得到一组连续填充三角形，形成的三角形序列为：(1, 2, 3)，(2, 3, 4)，(3, 4, 5)等，如图 5.1(a)所示。连续填充三角形扇与三角形类似，只是新顶点与上一个三角形的 1、3 顶点形成新的三角形，即始终保持一个顶点不变，形成的三角形序列为：(1, 2, 3)，(1, 3, 4)，(1, 4, 5)等，如图 5.1(b)所示。独立填充三角形也是由 glBegin 和 glEnd 之间的一系列顶点定义，所不同的是：每 3 个顶点定义一个三角形，每个三角形的顶点与它前面定义的三角形顶点

间没有任何关系，如图 5.1(c)所示。若 glBegin 和 glEnd 之间的顶点个数为 3n+k(k=0，1，2)，则定义了 n 个三角形，最后的 k 个顶点将被忽略。若 glBegin 与 glEnd 之间定义的顶点少于 3 个，则此定义不产生任何图元。

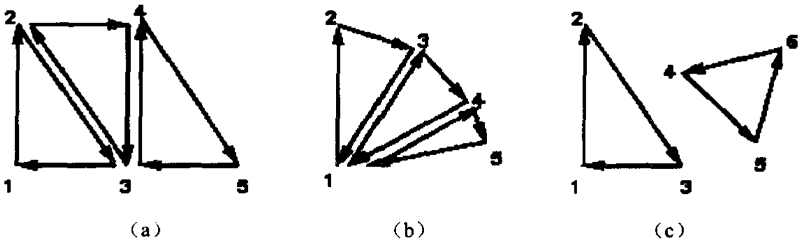

图5.1　3组三角形图元比较

连续填充四边形串定义了依次与前一个四边形共享一条边的四边形序列。假设 glBegin()和 glEnd()之间的顶点个数为 m，假定所定义的顶点序列为：v1，v2，v3，…，v_m。则此序列定义了$\lfloor m/2 \rfloor$-1(“$\lfloor\rfloor$”符号表示向下取整)个四边形。第 i 个四边形的 4 个顶点依次为：v2i.1，v2i，v2i+2，v_{2i+1}，i=1，…，$\lfloor m/2 \rfloor$-1，如图 5.2(a)所示。和连续填充三角形串一样，多余的顶点将被忽略。独立填充四边形与独立填充三角形类似，顺序的每 4 个顶点形成一个四边形，新四边形顶点与它前面定义的四边形顶点间没有任何关系，如图 5.2(b)所示。若 glBegin()与 glEnd()之间少于 4 个顶点，则不产生任何图元。

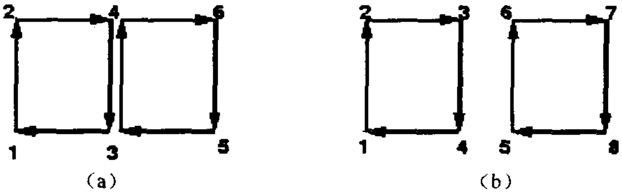

图5.2　2组四边形图元比较

5.2.2　特殊多边形的绘制方法

OpenGL 中定义的多边形只能是凸多边形，但在实际应用中往往需要绘制一些凹多边形。为了能在 OpenGL 中绘制出这类多边形，通常的办法是对它进行剖分，即将凹多边形剖分为一系列的凸多边形(其中三角形剖分最为常见)，例如，可以将以下左边的凹多边形剖分为右边的 4 个三角形，如图 5.3 所示。

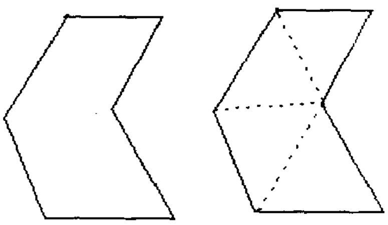

图5.3　凹多边形的三角形剖分

这样绘制左边的凹多边形就可以通过绘制右边的 4 个三角形来替代。显然，绘制时三角形的有些边不应该产生绘制，否则，多边形内就会出现多余的线框。OpenGL 中提供了解决这个问题的方法，可以通过设置边标志命令 glEdgeFlag 来控制某些面产生绘制，而另外一些面不产生绘制，亦可称之为标志边界线和非边界线。该命令的定义如下：

```
void glEdgeFlag(GLboolean flag); 或
void glEdgeFlagv(PGLboolean pflag);
```

当 flag 取 GL_TRUE 时，设置为边界线边；当 flag 取 GL_FALSE 时，设置为非边界线边。

5.2.3 点、线与多边形属性的设置

通常，屏幕上的点被绘成单个像素，线被绘成一个像素宽的实线，多边形绘成实心填充多边形。而在实际中常常遇到其他情况，以下就来讨论如何修改这些默认的设置。

设置顶点绘制大小的命令是 void glPointSize(GLfloat size)，它用来设置以像素为单位的顶点绘制时的宽度。

size 用来指定点的宽度，即像素宽度，其值必须大于 0.0，默认大小为 1.0。实际绘制时，其大小受反走样状态设置情况影响。在关闭反走样的情况下，设置的非整数宽度被截断为整数值。例如，当设置宽度 2.3，实际的绘制宽度将为 2；当设置宽度 3.7，实际宽度为 3。在激活了反走样的情况下，则非整数宽度不被截断。在绘制一组像素的边界上用较低亮度绘出，这样边缘就会显得平滑一些。

OpenGL 对允许的最大点大小有限制，设置为大于此值的数将自动截断为最大点的大小。不同的 OpenGL 版本对最大点大小有不同的规定，可以调用 glGetFloatv 命令，将参数设置为 GL_POINT_SIZE_RANGE，可以得到返回的当前版本所允许的最大点大小的值。

线属性比点属性稍微复杂一些，分为线宽和线型 2 种属性。OpenGL 调用下面的命令来设置线宽：

```
void glLineWidth(GLint size);
```

该命令用来指定以像素为单位的线宽度。

size 指定线宽，其值必须大于 0.0，默认值为 1.0。线的绘制与点的绘制一样，实际绘制的线宽一样受到反走样状态设置情况的影响。当反走样状态未被设置时，指定 2.1，1.8，3.5 大小的线宽将分别被绘成 2，1，3 个像素宽。当设置了反走样状态时，边界像素将会被部分填充。反走样的实际绘制结果是：以精确线为中心轴，绘制了一个给定宽度的填充矩形。

同样 OpenGL 对允许绘制的最大线宽也有限制。调用 glGetFloatv 过程，将参数设为 GL_LINE_WIDTH_RANGE，可以得到 OpenGL 允许的最大线宽值。

除了线宽之外，线的另外一个属性是线型。OpenGL 中用以下命令来指定线型：

```
void glLineStipple(GLint factor,  GLshort pattern);
```

其中，factor 指定线型模式中每位的重复次数，factor 在 [1，255]之间取值，默认取值为 1。pattern 用 16 位整数指定位模式。某位为 1，则在相应位置要产生绘制；某位为 0

时，则在相应位置不产生绘制。默认情况下，每一位均为 1，这种情况下调用直线绘制将绘出一条实线。

前面曾经说过，OpenGL 是工作在一系列状态之下的，这里对线型的设置就涉及到一种状态。在调用了所需的线型设置过程之后，还需调用命令 glEnable(GL_LINE_STIPPLE) 激活线型状态。这样设置的线型才能真正起作用。与其他状态一样，若想关闭线型的作用，可以调用 glDisable(GL_LINE_STIPPLE)，这样，即使指定了线型，绘制的直线也仍然是一条实线。

多边形的属性，主要是它的填充模式，默认情况下，多边形绘制使用实心填充模式。同样，多边形也可以用其他填充模式进行绘制，所用的命令是：

```
void glPolygonStipple (PGLubyte mask);
```

其中：mask 是一个指定 32×32 点画模式的数组，当某位值为 1 时在相应位置绘制，否则在相应位置不绘制。mask 是由 128 个字符组成的数组，默认情况下，由 mask 控制的绘制次序为：自左而右，自下而上，且字节高位在先。但也可以用 glPixelStore 加以改变。

与线型设置一样，多边形属性设置状态也需要激活和关闭。调用的参数如下：

- glEnable(GL_POLYGON_STIPPLE);
- glDisable(GL_POLYGON_STIPPLE);

5.2.4　法向量的计算方法

对于一个平面，其上各点的法向量都一样，所以称为平面法向。而对于一个曲面，虽然在计算机图形学中常常用许多小面片去逼近，但事实上每个顶点的法向都不一样。对于曲面上的每个点的法向量计算就有不同的方法，可以根据不同的应用选用合适的方法进行计算。当然，所得到的结果也往往不尽相同。OpenGL 本身并未提供计算法向量的函数，但它提供了指定法向量的函数。计算法向量的工作交由程序员自己去完成，这样就使程序员有很大的灵活性。计算出法向量后，再将它指定给每个顶点即可。下面先简单介绍一下法向量的基本概念，再给出它的计算方法，最后指出如何在 OpenGL 中指定法向量。

法向量是面上某一点处垂直于该面的向量。在平展的面上，各点有同一个方向，因此有同样的法向量；而在弯曲的曲面上，各点具有不同的法向量。几何对象的法向量定义它在空间中的方向，特别定义相对于光源的方向，因此法向量决定对象的在顶点上可接受多少光照。

对法向量的计算通常分为解析曲面和一般多边形来求解。解析曲面是用数学方程或方程组描述的平滑可导曲面。在大多数情况下，解析曲面的法向量较为容易求解得到。假设解析曲面可由如下的显式形式定义：

$$p(u, v) = [\ x(u, v),\ y(u, v),\ z(u, v)\ ]$$

其中，u，v 被限制在某个定义域内，x，y，z 分别是 3 个定义在 u，v 上的可导函数。为了计算法向量，需要求得 $\partial p / \partial u$，$\partial p / \partial v$ 两项。它们分别是在 u，v 方向上与曲面相切的向量。两个向量的叉积，垂直于两个向量，也即垂直于该点处曲面的切平面，因此这两个向量的叉积就是曲面上该点的法向量。通常计算出法向量后，需除以其长度，

以得到单位向量，这时需注意可能会有向量长度为 0 的退化情况。

对于隐式描述的解析曲面 F(x，y，z) = 0，求解法向量较为困难。在某些情况下，可以解出一个变量，例如：z = G(x，y)，并转换为上述的显式形式，再根据上面的方法求得法向量。一般而言，对于隐式表示的曲面方程，在一个特定点(x， y， z)上可以用梯度来计算法向量，即计算$\nabla F = [\partial F / \partial x,\ \partial F / \partial y,\ \partial F / \partial z]$即可。

对于一般多边形，可以利用不位于一条直线上多边形的 3 个顶点 v1， v2， v3。计算叉积[v1-v2] × [v2 - v3]，再将其单位化，求得该多边形的单位法向量。如果希望得到多边形各个顶点的法向量，需将顶点周围多边形的法向量相加。如图 5.4 所示，P 点周围有 4 个多边形，假设它们的法向量分别为 n1，n2，n3，n4，可以先计算 n1+n2+n3+n4，再将其单位化，便得到 P 点的法向量。

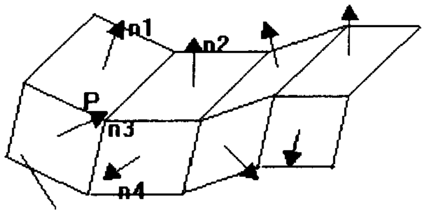

图5.4 曲面顶点法向量计算

5.2.5 利用简单几何图元构造较复杂形体

计算机图形学中的所有光滑曲面最终都是由多边形逼近得到的。如何用多边形来构造曲面，既需要有一定的经验积累，有时也需要有一些艺术头脑。造型是图形处理的基础，造型的好坏将直接影响其后续处理的质量和效果，为了能够用多边形较为理想地构造出几何形体，在利用多边形构造曲面多面体时应当注意以下几点：

- 保持多边形方向的一致性(通常，每个多边形均取逆时针方向)。
- 尽量选用三角形来逼近，这样可以保证多边形上的所有顶点共面。
- 避免造型过程中出现 T 型点(如图 5.5 左图所示)，因为 T 型点将导致绘制时出现裂缝。解决的办法是增加一条指向 T 型点的边(如图 5.5 右图所示)。
- 避免使用狭长三角形，应尽量选用各条边长度接近的三角形。狭长三角形将使绘制的结果颜色变化太尖锐。
- 要正确处理好图像质量和绘制速度的矛盾，因此要选用适当的多边形数目。

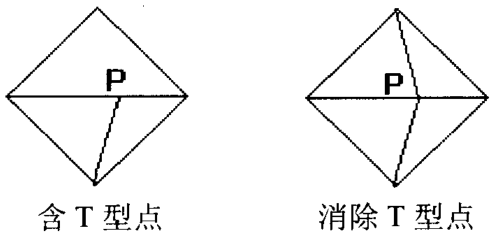

图5.5 T型点及其消除

下面通过用三角形构造球面的例子，展示如何用简单几何图元去构造复杂形体。显

然，可以用正多面体去逼近球，正多面体面数越多，它与球的逼近程度也就越高。首先，构造一个正二十面体。正二十面体有 12 个顶点，每个面是一个等边三角形。可以用如下的程序构造一个二十面体：

```
#define X 0.525731
#define Y 0.850651
static GLfloat vdata[12][3] = {
    {.x,  0.0,  z}, { x,  0.0,  z}, {.x,  0.0, .z}, { x,  0.0, .z},
    { 0.0,  z,  x}, { 0.0,  z, .x}, { 0.0, .z,  x}, { 0.0, .z, .x},
    { z,  x,  0.0}, {.z,  x,  0.0}, {.z,  x,  0.0}, {.z, .x,  0.0}
};
static GLfloat tindices[20][3] = {
    { 0,  4,  1}, { 0,  9,  4}, { 9,  5,  4}, { 4,  5,  8}, { 4,  8,
1},
    { 8, 10,  1}, { 8,  3, 10}, { 5,  3,  8}, { 5,  2,  3}, { 2,  7,
3},
    { 7, 10,  3}, { 7,  6, 10}, { 7, 11,  6}, { 7, 11,  6}, {11,  0,
6},
    { 6,  1, 10}, { 9,  0, 11}, { 9, 11,  2}, { 9,  2,  5}, { 7,  2,
11}
}

for ( i = 0;  i < 20;  i++){
    /* 请在此加入程序 */
    glBegin(GL_TRIANGLE);
    glVertex3fv(&vdata[tindices[i][0][0]);
    glVertex3fv(&vdata[tindices[i][1][0]);
    glVertex3fv(&vdata[tindices[i][2][0]);
    glEnd();
}
```

上面的程序中定义了特殊的数字 x 和 z，借助它们可以保证从原点到二十面体的任意一个顶点之间的距离均为 1.0。数组 vdata 给出了 12 个顶点的坐标，数组 tindices 指出如何连接顶点形成三角形。遵循上面提到的原则，这里所有的三角形均为逆时针方向。

为了增强绘制的立体感效果，可以为每个顶点指定法向量，同时激活光照。如果顶点处的法向量取所在面的法向量，则三角形 3 个顶点的法向量相等，因此只要指定一次即可。计算法向量的程序如下所示：

```
GLfloat n1[3],  n2[3],  n3[3],  norm[3];
for ( j = 0;  j < 3;  j++){
    n1[j] = vdata[tindices[i][0]][j] . vdata[tindices[i][1]][j];
    n2[j] = vdata[tindices[i][1]][j] . vdata[tindices[i][2]][j];
```

```
    }
    cross_product(n1, n2, norm);
    glNormal3fv(norm);
```

其中，cross_product 用于计算 n1，n2 叉乘结果的单位向量。将这段程序插入上面程序段中“请在此加入程序”处。同时，在调用这里的绘制程序之前，先指定好各种光照属性，从而使二十面体的绘制更富立体感。利用这种法向量计算方法求出的顶点法向量，在每个三角形面上都不相同，因此会产生不连续的感觉。通常解决这个问题的办法是，用周围平面法向量的平均值表示顶点的法向量。但是对于球上的顶点大可不必用这种方法，由于构造的是单位半径的球，因此球上顶点的法向量即为球心指向该点的矢量，又由于球心在原点，因此球上顶点的法向量与该点的坐标值是一致的。可以将上述程序段的相应地改成下面这样：

```
for ( i = 0; i < 20; i++){
    glBegin(GL_TRIANGLE);
    glNormal3fv(&vdata[tindices[i][0][0]);
    glVertex3fv(&vdata[tindices[i][0][0]);
    glNormal3fv(&vdata[tindices[i][1][0]);
    glVertex3fv(&vdata[tindices[i][1][0]);
    glNormal3fv(&vdata[tindices[i][2][0]);
    glVertex3fv(&vdata[tindices[i][2][0]);
    glEnd();
}
```

显然，用正二十面体来表示一个球显得过于粗糙，可以通过增加面数的办法使正多面体和球更为接近。增加正多面体有多种方法，一种简单的方法是剖分，即将前面定义的三角形面分成几个面，例如，一分为四，形成 4 个多边形。剖分的算法如下所示：

```
void subdivide(float *v1, float *v2, float *v3)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;
    for ( i = 0; i < 3; i++ ){
        v12[I] = v1[i]+v2[i];
        v23[I] = v2[i]+v3[i];
        v31[I] = v2[i]+v1[i];
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    drawtriangle(v1, v12, v31);
    drawtriangle(v2, v23, v12);
    drawtriangle(v3, v31, v23);
```

```
        drawtriangle(v12, v23, v31);
    }
```

其中，v1，v2，v3 表示三角形的 3 个顶点，normalize 和 drawtriangle 分别表示将向量单位化和根据 3 个顶点绘制三角形。

5.3 基本图元绘制实例程序

下面举例说明在 OpenGL 中绘制以上所指出的各种几何图元的主要源程序：

```
// oglshvw.cpp : CoglshapeView 类的实现
#include "stdafx.h"
#include "oglshape.h"

#include "oglshdoc.h"
#include "oglshvw.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

// COglshapeView
IMPLEMENT_DYNCREATE(COglshapeView, CView)

    // 以下定义了各种消息映射
BEGIN_MESSAGE_MAP(COglshapeView, CView)
    //{{AFX_MSG_MAP(COglshapeView)
    ON_WM_CREATE()
    ON_WM_DESTROY()
//  绘制 LINELOOP 的菜单项事件响应
    ON_COMMAND(ID_DRAW_LINELOOP, OnDrawLineloop)
    ON_UPDATE_COMMAND_UI(ID_DRAW_LINELOOP, OnUpdateDrawLineloop)
//  绘制 LINE 的菜单项事件响应
    ON_COMMAND(ID_DRAW_LINES, OnDrawLines)
    ON_UPDATE_COMMAND_UI(ID_DRAW_LINES, OnUpdateDrawLines)
//  绘制 LINESTRIP 的菜单项事件响应
    ON_COMMAND(ID_DRAW_LINESTRIP, OnDrawLinestrip)
    ON_UPDATE_COMMAND_UI(ID_DRAW_LINESTRIP                      ,
OnUpdateDrawLinestrip)
```

```
    // 绘制凹多边形的菜单项事件响应
        ON_COMMAND(ID_DRAW_NONCONVEX, OnDrawNonconvex)
        ON_UPDATE_COMMAND_UI(ID_DRAW_NONCONVEX
OnUpdateDrawNonconvex)
    // 绘制 POINTS 的菜单项事件响应
        ON_COMMAND(ID_DRAW_POINTS, OnDrawPoints)
        ON_UPDATE_COMMAND_UI(ID_DRAW_POINTS, OnUpdateDrawPoints)
    // 绘制 POLYGONS 的菜单项事件响应
        ON_COMMAND(ID_DRAW_POLYGONS, OnDrawPolygons)
        ON_UPDATE_COMMAND_UI(ID_DRAW_POLYGONS, OnUpdateDrawPolygons)
    // 绘制独立填充四边形的菜单项事件响应
        ON_COMMAND(ID_DRAW_QUADRILATERALS, OnDrawQuadrilaterals)
        ON_UPDATE_COMMAND_UI(ID_DRAW_QUADRILATERALS,
      OnUpdateDrawQuadrilaterals)
    // 绘制连续填充四边形串的菜单项事件响应
        ON_COMMAND(ID_DRAW_QUADSTRIP, OnDrawQuadstrip)
        ON_UPDATE_COMMAND_UI(ID_DRAW_QUADSTRIP
OnUpdateDrawQuadstrip)
    // 绘制三角形扇的菜单项事件响应
        ON_COMMAND(ID_DRAW_TRIANGLEFAN, OnDrawTrianglefan)
        ON_UPDATE_COMMAND_UI(ID_DRAW_TRIANGLEFAN,
          OnUpdateDrawTrianglefan)
    // 绘制独立填充三角形的菜单项事件响应
        ON_COMMAND(ID_DRAW_TRIANGLES, OnDrawTriangles)
        ON_UPDATE_COMMAND_UI(ID_DRAW_TRIANGLES
OnUpdateDrawTriangles)
    // 绘制连续填充三角形串的菜单项事件响应
        ON_COMMAND(ID_DRAW_TRIANGLESTRIP, OnDrawTrianglestrip)
        ON_UPDATE_COMMAND_UI(ID_DRAW_TRIANGLESTRIP
OnUpdateDrawTrianglestrip)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// COglshapeView 构造与析构函数
COglshapeView::COglshapeView()
{
    }

COglshapeView::~COglshapeView()
```

```
{
}

// COglshapeView 类的屏幕绘制响应函数
void COglshapeView::OnDraw(CDC* pDC)
{
        COglshapeDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
        wglMakeCurrent(pDC.>m_hDC, m_hRC);
    DrawWithOpenGL();
    wglMakeCurrent(pDC.>m_hDC, NULL);
}

// COglshapeView diagnostics
#ifdef _DEBUG
void COglshapeView::AssertValid() const
{
    CView::AssertValid();
}

void COglshapeView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

COglshapeDoc* COglshapeView::GetDocument() // 非测试版
{
    ASSERT(m_pDocument.>IsKindOf(RUNTIME_CLASS(COglshapeDoc)));
    return (COglshapeDoc*)m_pDocument;
}
#endif //_DEBUG

// COglshapeView 创建初始化
int COglshapeView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
        PIXELFORMATDESCRIPTOR pfd =
    {
```

```
        sizeof(PIXELFORMATDESCRIPTOR),    // 结构大小
        1,                                // 结构的版本号
        PFD_DRAW_TO_WINDOW |              // 属性标志
        PFD_SUPPORT_OPENGL,
        PFD_TYPE_RGBA,
        24,                               // 24 位颜色
        0, 0, 0, 0, 0, 0,                 // 无须关心的数据
        0, 0, 0, 0, 0, 0, 0,              // 不是透明度或累加器缓存
        32,                               // 32 位深度缓存
        0, 0,                             // 非模板或辅助缓存
        PFD_MAIN_PLANE,                   // 主层次类型
        0,                                // 保留项
        0, 0, 0
    };
    CClientDC clientDC(this);
    int pixelFormat = ChoosePixelFormat(clientDC.m_hDC, &pfd);
    BOOL success = SetPixelFormat(clientDC.m_hDC, pixelFormat,
&pfd);

    m_hRC = wglCreateContext(clientDC.m_hDC);

     return 0;
}

void COglshapeView::OnDestroy()
{
    CView::OnDestroy();
        wglDeleteContext(m_hRC);
}

BOOL COglshapeView::PreCreateWindow(CREATESTRUCT& cs)
{
        cs.style |= WS_CLIPCHILDREN | WS_CLIPSIBLINGS;
return CView::PreCreateWindow(cs);
}

    //  设置待绘制几何图元为 LineLoop
    void COglshapeView::OnDrawLineloop()
{
```

```
    m_shape = LineLoop;
    Invalidate();
}

    //  表明当前几何图元为 LineLoop
void COglshapeView::OnUpdateDrawLineloop(CCmdUI* pCmdUI)
{
        if (m_shape == LineLoop)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawLines()
{
        m_shape = Lines;
    Invalidate();
}

void COglshapeView::OnUpdateDrawLines(CCmdUI* pCmdUI)
{
        if (m_shape == Lines)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawLinestrip()
{
        m_shape = LineStrip;
    Invalidate();
}

void COglshapeView::OnUpdateDrawLinestrip(CCmdUI* pCmdUI)
{
        if (m_shape == LineStrip)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
```

```
}

void COglshapeView::OnDrawNonconvex()
{
        m_shape = Nonconvex;
    Invalidate();
}

void COglshapeView::OnUpdateDrawNonconvex(CCmdUI* pCmdUI)
{
        if (m_shape == Nonconvex)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawPoints()
{
        m_shape = Points;
    Invalidate();
}

void COglshapeView::OnUpdateDrawPoints(CCmdUI* pCmdUI)
{
        if (m_shape == Points)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawPolygons()
{
        m_shape = Polygons;
    Invalidate();
}

void COglshapeView::OnUpdateDrawPolygons(CCmdUI* pCmdUI)
{
        if (m_shape == Polygons)
```

```
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawQuadrilaterals()
{
        m_shape = Quads;
    Invalidate();
}

void COglshapeView::OnUpdateDrawQuadrilaterals(CCmdUI* pCmdUI)
{
        if (m_shape == Quads)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawQuadstrip()
{
        m_shape = QuadStrip;
    Invalidate();
}

void COglshapeView::OnUpdateDrawQuadstrip(CCmdUI* pCmdUI)
{
        if (m_shape == QuadStrip)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawTrianglefan()
{
        m_shape = TriangleFan;
    Invalidate();
}
```

```
void COglshapeView::OnUpdateDrawTrianglefan(CCmdUI* pCmdUI)
{
	if (m_shape == TriangleFan)
		pCmdUI->SetCheck(TRUE);
	else
		pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawTriangles()
{
	m_shape = Triangles;
	Invalidate();
}

void COglshapeView::OnUpdateDrawTriangles(CCmdUI* pCmdUI)
{
	if (m_shape == Triangles)
		pCmdUI->SetCheck(TRUE);
	else
		pCmdUI->SetCheck(FALSE);
}

void COglshapeView::OnDrawTrianglestrip()
{
	m_shape = TriangleStrip;
	Invalidate();
}

void COglshapeView::OnUpdateDrawTrianglestrip(CCmdUI* pCmdUI)
{
	if (m_shape == TriangleStrip)
		pCmdUI->SetCheck(TRUE);
	else
		pCmdUI->SetCheck(FALSE);
}

	// 根据菜单项的选择绘制不同的图元
void COglshapeView::DrawWithOpenGL()
{
```

```
    // 首先进行清除窗口的操作
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

        // 指定颜色为黑色
    glColor3f(0.0f, 0.0f, 0.0f);

    switch(m_shape)
    {
        case Points        : DrawPoints();  break;
        case Lines         : DrawLines();  break;
        case LineStrip     : DrawLineStrip();  break;
        case LineLoop      : DrawLineLoop();  break;
        case Polygons      : DrawPolygons();  break;
        case Triangles     : DrawTriangles();  break;
        case TriangleStrip : DrawTriangleStrip();  break;
        case TriangleFan   : DrawTriangleFan();  break;
        case Quads         : DrawQuads();  break;
        case QuadStrip     : DrawQuadStrip();  break;
        case Nonconvex     : DrawNonconvex();  break;
    }

    // 迫使屏幕立即产生绘制
    glFlush();
}

// 绘制点的函数，绘制出 OpenGL 允许的最大点
void COglshapeView::DrawPoints()
{
    GLfloat ps[2];
    glGetFloatv(GL_POINT_SIZE_RANGE, ps);

    glPointSize(ps[1]);

    glBegin(GL_POINTS);
        glVertex2f(0.0f, 0.0f);
        glVertex2f(0.75f, 0.75f);
        glVertex2f(-0.75f, -0.75f);
        glVertex2f(-0.75f, 0.75f);
```

```
        glVertex2f(0.75f, -0.75f);
    glEnd();
}

// 绘制线的函数，包括不同的线宽、线型
void COglshapeView::DrawLines()
{
    GLfloat lw[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE, lw);

    if (lw[1] >= 4.0f)
        glLineWidth(4.0f);
    else
        glLineWidth(lw[1]);

    glBegin(GL_LINES);

        glVertex2f(-0.75f, 0.75f);
        glVertex2f(0.75f, 0.75f);

        glVertex2f(-0.75f, 0.25f);
        glVertex2f(0.75f, 0.25f);

    glEnd();

    glLineStipple(1, 0x4444);
    glEnable(GL_LINE_STIPPLE);

    glBegin(GL_LINES);

        glVertex2f(-0.75f, .0.25f);
        glVertex2f(0.75f, -0.25f);

        glVertex2f(-0.75f, -0.75f);
        glVertex2f(0.75f, -0.75f);

    glEnd();

    glDisable(GL_LINE_STIPPLE);
```

```
}

   //  绘制 LINESTRIP 的函数
void COglshapeView::DrawLineStrip()
{
    GLfloat lw[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE,  lw);

    if (lw[1] >= 4.0f)
        glLineWidth(4.0f);
    else
        glLineWidth(lw[1]);

    glBegin(GL_LINE_STRIP);

        glVertex2f(-1.00f,  0.75f);
        glVertex2f(1.0f,  0.75f);

        glVertex2f(-1.0f,  0.25f);
        glVertex2f(1.0f,  0.25f);

        glVertex2f(-1.0f,  -0.25f);
        glVertex2f(1.0f,  -0.25f);

        glVertex2f(-1.0f,  -0.75f);
        glVertex2f(1.0f,  -0.75f);

    glEnd();
}

//  绘制 LINELOOP 的函数
void COglshapeView::DrawLineLoop()
{
    GLfloat lw[2];
    glGetFloatv(GL_LINE_WIDTH_RANGE,  lw);
    if (lw[1] >= 4.0f)
        glLineWidth(4.0f);
    else
        glLineWidth(lw[1]);
```

```
    glBegin(GL_LINE_LOOP);
        glVertex2f(-1.00f, 0.75f);
        glVertex2f(1.0f, 0.75f);

        glVertex2f(-1.0f, 0.25f);
        glVertex2f(1.0f, 0.25f);

        glVertex2f(-1.0f, -0.25f);
        glVertex2f(1.0f, -0.25f);

        glVertex2f(-1.0f, -0.75f);
        glVertex2f(1.0f, -0.75f);
    glEnd();
}

// 绘制多个多边形的函数，包括不同的填充模式
void COglshapeView::DrawPolygons()
{
    glLineWidth(1.0f);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_POLYGON);
        glVertex2f(.0.5f, 0.75f);
        glVertex2f(-0.75f, 0.27f);
        glVertex2f(-0.5f, 0.10f);
        glVertex2f(0.5f, 0.10f);
        glVertex2f(0.75f, 0.27f);
        glVertex2f(0.5f, 0.75f);
    glEnd();

    // 指定的多边形填充模式
    GLubyte pattern[] =
    {
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0,
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0,
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0,
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
```

```
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0,
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0,
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0,
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0,
        0xff, 0x00, 0xff, 0x00, 0x0f, 0xf0, 0x0f, 0xf0,
        0x00, 0xff, 0x00, 0xff, 0x0f, 0xf0, 0x0f, 0xf0
    };

    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glPolygonStipple(pattern);
    glEnable(GL_POLYGON_STIPPLE);

    glBegin(GL_POLYGON);
        glVertex2f(-0.5f, -0.10f);
        glVertex2f(-0.75f, -0.27f);
        glVertex2f(-0.5f, -0.75f);
        glVertex2f(0.5f, -0.75f);
        glVertex2f(0.75f, -0.27f);
        glVertex2f(0.5f, -0.10f);
    glEnd();

    glDisable(GL_POLYGON_STIPPLE);
}

// 绘制凹多边形的函数，其中用的就是标志多边形边的方法
void COglshapeView::DrawNonconvex()
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    glBegin(GL_POLYGON);
        glEdgeFlag(TRUE);
        glVertex2f(-0.2f, 0.3f);
        glEdgeFlag(FALSE);
        glVertex2f(-0.2f, -0.1f);
        glEdgeFlag(TRUE);
        glVertex2f(0.0f, -0.1f);
```

```
        glVertex2f(0.0f, 0.3f);
    glEnd();

    glBegin(GL_POLYGON);
        glEdgeFlag(TRUE);
        glVertex2f(-0.2f, -0.1f);
        glVertex2f(-0.2f, -0.4f);
        glEdgeFlag(FALSE);
        glVertex2f(0.0f, -0.4f);
        glVertex2f(0.0f, -0.1f);
    glEnd();

    glBegin(GL_POLYGON);
        glEdgeFlag(FALSE);
        glVertex2f(0.0f, -0.1f);
        glEdgeFlag(TRUE);
        glVertex2f(0.0f, -0.4f);
        glVertex2f(0.3f, -0.4f);
        glVertex2f(0.3f, -0.1f);
    glEnd();

    glEnd();
}

// 绘制多个独立填充三角形的函数
void COglshapeView::DrawTriangles()
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_TRIANGLES);
        glVertex2f(-0.5f, 0.3f);
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(0.5f, -0.5f);

        glVertex2f(-0.5f, 0.8f);
        glVertex2f(-0.5f, 0.5f);
        glVertex2f(0.5f, 0.5f);
    glEnd();
}
```

```
// 绘制连续填充三角形序列
void COglshapeView::DrawTriangleStrip()
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_TRIANGLE_STRIP);
        glVertex2f(-0.75f, 0.25f);
        glVertex2f(-0.75f, -0.25f);
        glVertex2f(-0.25f, 0.25f);

        glVertex2f(0.0f, -0.25f);

        glVertex2f(0.25f, 0.5f);

        glVertex2f(0.50f, 0.3f);

        glVertex2f(0.75f, 0.3f);
    glEnd();
}

// 绘制三角形扇的函数
void COglshapeView::DrawTriangleFan()
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_TRIANGLE_FAN);
        glVertex2f(-0.25f, 0.25f);
        glVertex2f(-0.25f, -0.25f);
        glVertex2f(0.25f, 0.25f);

        glVertex2f(0.25f, 0.5f);

        glVertex2f(-0.05f, 0.7f);

        glVertex2f(-0.45f, 0.5f);
    glEnd();
}

// 绘制多组独立填充四边形的函数
void COglshapeView::DrawQuads()
{
```

```
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_QUADS);
        glVertex2f(-0.75f, 0.75f);
        glVertex2f(-0.75f, 0.25f);
        glVertex2f(-0.25f, 0.0f);
        glVertex2f(0.0f, 0.5f);

        glVertex2f(-0.55f, -0.25f);
        glVertex2f(-0.75f, .0.45f);
        glVertex2f(-0.25f, .0.7f);
        glVertex2f(0.5f, -0.25f);

        glVertex2f(0.5f, 0.75f);
        glVertex2f(0.25f, 0.1f);
        glVertex2f(0.75f, 0.2f);
        glVertex2f(0.8f, 0.65f);
    glEnd();
}

// 绘制连续填充四边形序列的函数
void COglshapeView::DrawQuadStrip()
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glBegin(GL_QUAD_STRIP);
        glVertex2f(-0.75f, 0.25f);
        glVertex2f(-0.65f, 0.75f);
        glVertex2f(-0.55f, 0.1f);
        glVertex2f(-0.25f, 0.6f);

        glVertex2f(-0.15f, -0.2f);
        glVertex2f(0.0f, 0.45f);

        glVertex2f(0.25f, -0.7f);
        glVertex2f(0.5f, 0.65f);

        glVertex2f(0.75f, -0.3f);
        glVertex2f(0.85f, 0.45f);
    glEnd();
}
```

第 6 章　坐标变换

坐标变换在图形绘制程序中的地位相当于微积分在高等数学中的作用，也就是说它可以看作是一切图形绘制程序的基础，几乎任何图形绘制程序都必须不同程度地依赖于坐标变换。本章将从以下几方面对坐标变换进行深入讲解：

- 通过在三维空间中进行变换实现从任意角度观看一个几何体。
- 控制模型在三维空间中的观看位置。
- 裁剪模型超出场景之外的部分。
- 操作适当的矩阵堆栈控制模型变换来进行绘制，并将模型投影到屏幕。
- 综合使用多种变换实现较复杂的动画，将给出一个详细的机器人手臂的例子。

6.1　坐标变换的基本概念

前面已经了解了如何利用 OpenGL 命令绘出建立在场景中的几何模型，要准确、灵活地绘制三维场景，还需了解如何将模型置于场景中，并选择合适的位置来观察场景。

三维几何物体是在三维坐标中考虑的，但当计算机图形的点绘制到屏幕上时，它将是二维图像。因此，将几何物体的三维坐标转换到屏幕上的像素位置，需要经过如下 3 种计算机操作：

- 通过矩阵相乘所表示的变换，包括造型、视图和投影等操作。这些操作包括旋转、平移、缩放、反射、正交投影和透视投影。通常需要将几种变换结合起来绘制场景。
- 由于场景是在一个矩形窗口中绘制的，所以位于窗口之外的物体(或物体的一部分)必须被裁剪掉。在三维计算机图形中，在裁剪面构成的范围以外的物体也被裁剪。
- 在变换的坐标和屏幕像素之间必须建立对应关系。这种操作称为视口变换。

为了解释清楚变换的基本概念，人们通常用照相机做类比。变换过程产生视图场景的过程也同使用照相机的步骤非常类似，(如图 6.1 所示)：

(1) 竖起三角架，将照相机对准场景(相当于视图变换)。

(2) 将要拍摄的场景置于所要求的位置(相当于造型变换)。

(3) 选择照相机透镜或调整焦距(相当于投影变换)。

(4) 确定最后的照片需要多大，例如照片放大(相当于视口变换)。

在完成上述这些步骤之后，就能进行照片的拍摄了，同样，在计算机中也就可以绘制三维图形了。

图6.1 变换的照相机类比

这些使用照相机的步骤对应于程序中所指定变换的次序，但在被操作物体的顶点上所进行的数学操作则不必按此顺序执行。在程序中，视图变换必须在造型变换之前完成，但投影变换和视口变换可以在绘图之前的任何时候指定。图 6.2 表示了在计算机中进行这些操作的顺序。

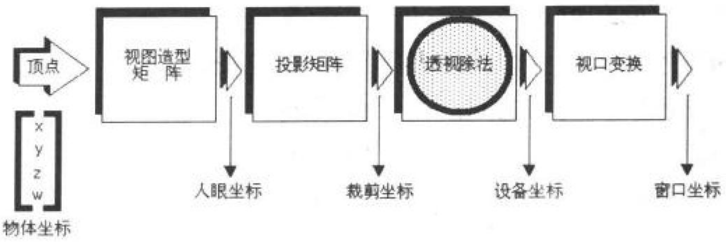

图6.2 顶点变换的阶段

(1) 指定视图和造型变换，把它们结合在一起构成 1 个视图变换。这个变换将输入的顶点从物体坐标变换到人眼坐标。

(2) 指定投影变换，使物体从人眼坐标变换到裁剪坐标。这个变换定义 1 个视图体，在视图体外的物体将被裁剪掉。

(3) 通过对坐标值除以 w，执行透视变换，变换到设备坐标。

(4) 通过视口变换将坐标变换到坐标。

这些变换仅对 x，y 执行坐标操作就可以确定在屏幕上要绘出的图像，但所有这些变换同时还要对 z 坐标执行。z 值恰当地反映出顶点的深度(即距离屏幕的远近)，此深度值还可用于消除隐藏面。

按上述步骤绘制 1 个立方体，首先用造型变换缩放这个立方体，然后所用的视图变换是沿着 z 轴的简单平移，另外还指定了投影和视口变换，如例 3.1 所示。

例 3.1　变换立方体

```
# include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void display (void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glLoadIdentity ();             /* 装入单位矩阵 */
   glTranslatef (0.0, 0.0, -5.0); /* 视图变换 */
   glScalef (1.0, 2.0, 1.0);      /* 模型变换 */
   auxWireCube(1.0);    /* 画立方体 */
   glFlush();
}
void myinit (void)
{
   glShadeModel (GL_FLAT);
}
void myReshape(GLsizei w, GLsizei h)
{
   glMatrixMode (GL_PROJECTION);    /* 为设置投影矩阵作准备 */
   glLoadIdentity ();               /* 定义投影矩阵 */
   glFrustum (-1.0, 1.0, -1.0, 1.0, /* 进行变换 */
              1.5, 20.0);
   glMatrixMode (GL_MODELVIEW);   /* 返回模型矩阵 */
   glViewport (0, 0, w, h);       /* 定义视口 */
}
```

```
int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
   auxInitPosition (0, 0, 500, 500);
   auxInitWindow (argv[0]);
   myinit ();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
}
```

6.2 通用变换命令

OpenGL 中用于指定变换时主要有以下 4 条命令:

```
glMatrixMode();
glLoadIdentity();
glLoadMatrix();
glMultiMatrix();
```

下面对上述 4 条命令逐一作介绍:

```
void glMatrixMode(GLenum mode);
```

此函数用于指定用哪 1 个矩阵作为当前矩阵。其中参数 mode 用于指定哪 1 种矩阵栈是其后矩阵操作的目标。mode 可以指定下面 3 个值:

- GL_MODELVIEW 把后面的矩阵操作施加于造型视图矩阵栈。
- GL_PROJECTION 把后面的矩阵操作施加于投影矩阵栈。
- GL_TEXTURE 把后面的矩阵操作施加于纹理矩阵栈。

一次只能修改一种矩阵。默认情况下矩阵操作施加于造型视图矩阵栈。

```
void glLoadIdentity(void);
```

将当前矩阵赋值为单位矩阵。

```
void glLoadMatrix{fd}(const TYPE *m);
```

将矩阵 m 赋值给当前矩阵，m 中包含矩阵的 16 个值，列优先组成 4×4 矩阵。

```
void glMultMatrix{fd}(const TYPE *m);
```

用任意矩阵乘以当前矩阵，m 的含义同上。例如，假设 m 由(m1，m2，…，m16)16 个值组成，则 m 所表示的矩阵 M 为:

$$M = \begin{bmatrix} m1 & m5 & m9 & m13 \\ m2 & m6 & m10 & m14 \\ m3 & m7 & m11 & m15 \\ m4 & m8 & m12 & m16 \end{bmatrix}$$

由于在用 C 语言编程时，如果说明 1 个矩阵为 m[4][4]，则 m[i][j]在 C 语言中表示第 i 行第 j 列的数据，而 OpenGL 则表示第 j 行第 i 列的数据，两者不相符。为避免混淆，在用

OpenGL 编程时应将矩阵说明为 m[16]。另外，glMultMatrix 命令中的矩阵乘法是右乘，即，若当前矩阵为 C，glMultMatrix 命令的参数矩阵为 M，则相乘的结果为 CM。OpenGL 中其他用到矩阵乘法的命令也都是如此。

6.3 视图和造型变换

在 OpenGL 中视图和造型变换的关系十分密切，实际上 2 个变换结合成一个矩阵 modelview(造型视图矩阵)。由于视图和造型变换具有二重性，所以要考虑 2 种类型变换的效果。用照相机类比在某个方向上移动照相机(用视图变换)，或在相反方向上移动物体(用造型变换)，根据特定的应用可以灵活地采用变换。

通过 6.2“通用变换命令”一节的介绍，在进行造型和视图变换之前，必须调用：

```
glMatrixMode(GL_MODEL_VIEW);
```

6.3.1 变换的顺序

当执行变换 A 和变换 B 时，如果按不同的顺序执行，得到的结果往往不同，变换的顺序是很重要的。例如，变换 A 为旋转 45° 角，变换 B 为在 x 轴正方向移动一个距离，不同的执行顺序产生如图 6.4 的不同结果。

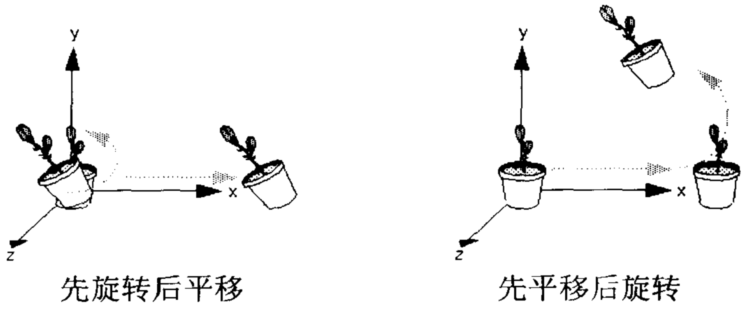

图6.4　不同变换顺序产生不同的结果

全部视图和造型变换表示为 4×4 矩阵。1 个新的 4×4 矩阵 M 乘当前造型视图矩阵 C，产生 CM，当前的造型视图矩阵再乘顶点 v，产生 CMv。上述过程说明，程序中绘制顶点前的最后一个变换命令最先作用于顶点之上。

例如，下面利用 3 个变换绘制单个顶点的命令序列。

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);                    /* 乘以矩阵 N */
glMultMatrixf(M);                    /* 乘以矩阵 M */
glMultMatrixf(L);                    /* 乘以矩阵 L */
glBegin(GL_POINTS);
glVertex3f(v);                       /* 画变换后的顶点 v */
```

```
glEnd();
```

在这个程序中，造型视图矩阵相继含有 I，N，NM，NML 矩阵，这里 I 表示单位矩阵。变换后的顶点为 NMLv。因此，顶点的变换为 N(M(Lv))，即先以 L 乘 v，而后以 M 乘 Lv，最后以 N 乘 MLv。这里，顶点 v 实际的变换顺序与指定的顺序相反。在变换的实现中实际上是单个造型视图矩阵与顶点相乘，N，M 和 L 矩阵在与顶点 v 相乘之前已相乘转换为单一矩阵。

变换有两种方法：

第 1 种方法是利用全局的固定坐标系统，必须考虑到矩阵相乘要按出现在程序中的相反顺序执行。再以旋转和平移为例，假如要求所绘制的物体在操作之后出现在轴上，则必须先旋转，后平移。这里，程序应写为(其中 R 和 T 分别为旋转和平移矩阵)：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T);                    /* 平移变换 */
glMultMatrixf(R);                    /* 旋转变换 */
draw_the_object();
```

第 2 种方法是设想 1 个局部坐标系统与所绘制的物体联系起来，全部操作的发生相对于这个变化的坐标系统，程序中矩阵的相乘按自然顺序出现。用平移旋转的例子来说明，以设想与某个坐标系统相联系的物体开始，沿 x 轴移动物体及其坐标系统。而后，绕已经平移的原点旋转，因此物体在 x 轴的该位置旋转。

这种方法可用于绘制机器人手臂上的肩、肘、腕和每个手指的接合。绘制从手臂的肩到腕和手指的每个接合处需要利用合适的旋转和平移。利用第 1 种方法，按逆向考虑变换是很容易出错的。但当非均匀的缩放与旋转混合使用时，利用第 2 种方法会造成局部坐标系统的轴成为非正交的。

6.3.2 造型变换

造型变换有以下 3 条 OpenGL 命令：

glTranslate*()：　平移

glRotate*()：　旋转

glScale*()：　缩放

OpenGL 自动计算这 3 条命令。这些命令的作用等价于调用 glMultMatrix*()，参数设置为相应的矩阵。但前者计算比后者快。

1. 平移

```
void glTranslate{fd}(TYPE x, TYPE y, TYPE z);
```

用平移矩阵乘当前矩阵。参数 x，y，z 分别指定 x，y，z 坐标值平移物体(或用相同的量移动局部坐标系统)。

图 6.5 表示了 glTranslate()的效果。

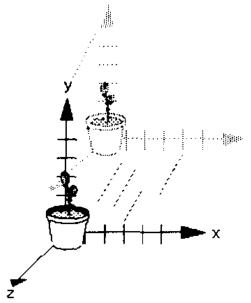

图6.5 物体平移

当 glTranslate()的参数设置为(0.0，0.0，0.0)时，对物体及其局部坐标系不起作用。

2. 旋转

```
void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
```

用旋转矩阵乘当前矩阵。其中 angle 指定旋转的角度(以度数为单位)，参数 x，y，z 指定一个从原点到(x，y，z)点的向量，作为旋转的中心轴，旋转的方向是逆时针方向。例如，glRotatef(45.0，0.0，0.0，1.0)的作用是绕 z 轴逆时针旋转 45 度角，如图 6.6 所示。

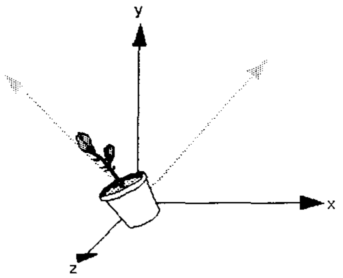

图6.6 物体旋转

离旋转轴较远的物体比近的物体有较大的旋转轨道。同样，当 glRotate*()命令的 angle 参数设置为 0 时，不产生任何作用。

3. 缩放

```
void glScale{fd}(TYPE x, TYPE y, TYPE z);
```

用缩放矩阵乘当前矩阵。x，y，z 依次指定按 x，y，z 轴各自的比例因子缩放物体(即伸展或收缩局部坐标轴)。

glScale(2.0，-0.5，1.0)的作用如图 6.7 所示。

glScale*()是 3 种造型变换中唯一改变物体实际大小的变换。用大于 1.0 的因子作为参数进行缩放变换的结果是将物体拉长，用小于 1.0 而大于 0 的因子作为参数进行缩放变换的结果是将物体缩短。用负数作为因子的结果是将物体对轴作反射。利用 glScale*()会降低光照计算的效果，因为在变换后法向量需要重新单位化。下面是一个造型变换的例子，在这个程序中 4 次绘制三角形。

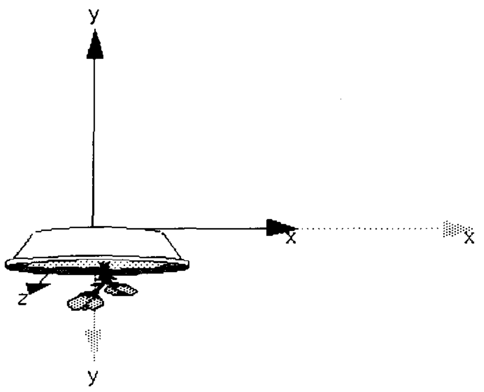

图6.7　物体缩放

(1) 不用造型变换绘制 1 个实线框三角形。
(2) 对三角形进行平移后，再用虚线框绘制三角形。
(3) 将三角形的高缩为一半，宽放为 1.5 倍，再用长虚线绘制三角形。
(4) 旋转三角形并用点线方式绘制三角形。
程序如下：

```
glLoadIdentity();
glColor3f(1.0, 1.0, 1.0);
draw_triangle();                          /* 实线 */
glEnable(GL_LINE_STIPPLE);          /* 虚线 */
glLineStipple(1, 0xF0F0);
glLoadIdentity();
glTranslatef(.20.0, 0.0, 0.0);
draw_triangle();

glLineStipple(1, 0xF00F);               /* 长间隔虚线 */
glLoadIdentity();
glScalef(1.5, 0.5, 1.0);
draw_triangle();

glLineStipple(1, 0x8888);               /* 点画线 */
glLoadIdentity();
glRotatef (90.0, 0.0, 0.0, 1.0);
draw_triangle ();
glDisable (GL_LINE_STIPPLE);
```

程序结果如图 6.8 所示。

图6.8　造型变换的结果

glLoadIdentity 起到了分隔造型变换的作用，这种初始化矩阵值的方法可以避免由于连续的变换而出现结果的累积，使得真正的造型操作指定在模型上。后面将在 6.6“操作矩阵栈”一节中详细讨论分隔变换的更好的方法。

有时希望在程序中连续地旋转一个物体，一般使用反复地将 1 个小的旋转矩阵施加在物体上的方法来实现。这种实现方法存在一个问题，由于舍入误差的存在，连续多次使用该旋转矩阵往往会偏离想要的结果，甚至有时会产生相反的结果。为了实现旋转，同时避免上面所说的弊端，可以定义一个旋转角度，每次递增该角度并用该角度执行一个新的旋转命令(每次调用命令前先对当前矩阵进行初始化)。

6.3.3　视图变换

视图变换改变视点的位置和方向。用照相机作类比，视图变换相当于定位照相机的三脚架，并将镜头指向要拍摄的场景。为了在 1 个特定的场景中组织最终的图像或照片，既可以移动相机，也可以沿着相反方向移动物体，这两者的效果是等价的。同样，按逆时针方向旋转物体的造型变换等价于按顺时针方向旋转照相机的视图变换。视图变换命令必须在任何造型变换命令执行之前调用，才能使造型变换首先作用于物体。

实现视图变换可以利用下面的几种方法之一，也可以通过选择视点的默认位置和方向来实现，即在原点沿着 z 轴的负半轴方向观察。

- 利用一个或几个造型变换命令(即 glTranslate*()和 glRotate*())。这些变换的结果可以看成是移动照相机的位置，也可看作移动相对于照相机静止的全部物体的位置。
- 利用实用库例程 gluLookAt()来定义视线。这个例程中封装了一系列的旋转和平移命令。
- 创建封装旋转和平移命令的实用例程。有些应用需要用简便的方法指定视图变换的定制例程。例如，在飞机飞行中指定翻动、俯仰和航向旋转角，或者有时对绕物体旋转的照相机指定一种极坐标。
- 利用 glTranslate*() 和 glRotate*()函数。

当使用造型变换命令来模仿视图变换(将视点移动到一个需要的位置，而物体本身保持不动)时，由于视点最初在原点上，而物体也最容易在原点上构造(如图 6.9 所示)，所以通常为了使物体可见，必须执行变换。图 6.9 中照相机初始对着 z 轴的负半轴。

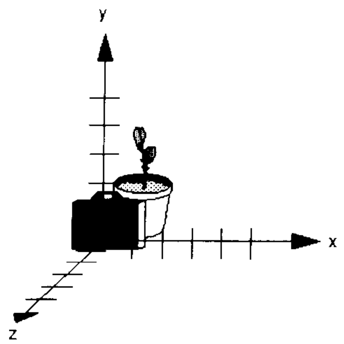

图6.9 在原点绘制的物体及其视点

在最简单的情况下，可以将视点向后移动离开物体，这与物体向前移动离开视点具有相同的效果。默认时向前移动是沿着 z 轴的负半轴。假如旋转视点，向前的含义就不同了。下面的命令是通过移动视点在视点和物体之间产生 5 个单位的距离，结果如图 6.10 所示。

```
glTranslatef(0.0, 0.0, -5.0);
```

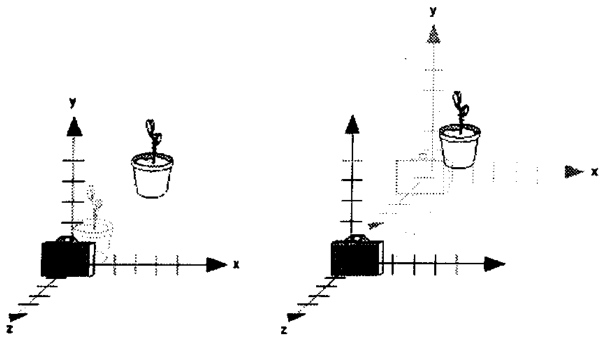

图6.10 将视点和物体分开

假设要从侧面观察物体，如果用全局和固定的坐标系统为基准，则物体和照相机都假设在原点。先旋转物体，然后把物体移开照相机，使所要的侧面变为可见。由于用的是固定坐标系统的方法，命令必须按起作用相反的顺序发出，所以在程序中应先写平移命令，然后写旋转命令。

现在考虑用局部坐标系统的方法。此时移动物体及其局部坐标系统离开原点，利用已平移后的坐标系统执行旋转命令。由于是按应用的顺序发出命令，因而同样平移在先。产生所要结果的变换命令如下：

```
glTranslatef(0.0,  0.0, -5.0);
glRotatef(90.0,  0.0,  1.0,  0.0);
```

应该注意的是，利用固定坐标系统，旋转总是发生在绕全局系统的原点，而利用局部坐标系统，旋转发生在绕局部系统的原点。此外，视图变换还可利用下面描述的 gluLookAt() 实用例程。

● 利用实用例程 gluLookAt()。

```
gluLookAt(GLdouble eyex,  GLdouble eyey,  GLdouble eyez,
GLdouble centerx,  GLdouble centery,  GLdouble centerz,
```

```
GLdouble upx, GLdouble upy, GLdouble upz);
```

在编程中经常先在原点或其他方便的位置创建场景，然后在任意一点观察它，以获得较好的观察效果。gluLookAt()实用例程正如其函数名所指出的，就是基于上述目的。此函数中用到 3 个参数，分别指定视点的位置、参照点的位置以及指定哪个向量为向上的方向。先选择一个视点来产生场景的所需视图。参照点通常是场景的中心点，例如假设在原点创建场景，参照点可能就是原点。指定正确的向上方向的向量可能需要有一些技巧。如果已经在原点或原点附近创建了一个真实世界场景，并且已经使-y 轴方向朝上，那么 gluLookAt()中用到的向上向量就是-y 轴。如果创建的是一个飞行仿真场景，向上向量则是与飞机机翼垂直的方向。

注意：gluLookAt()是实用库例程中的一个函数，而不是基本 OpenGL 库中的函数。这并不是说它没有用，而是因为它实际上是封装了几个 OpenGL 命令实现的，特别是 glTranslate*()命令和 glTranslate*()命令。gluLookAt()生成一个由视点位置，场景中心的参考点和向上向量所导出的视图矩阵。这个矩阵将视点转换为原点，将参照点转换到负 z 轴。视点和参照点连成视线，形成观察的方向，而向上向量则指出视图体从底到顶的方向，通常向上向量被置换到正 y 轴。但向上向量不一定平行于从视点到参照点的视线。

6.4 投影变换

本节将说明如何将定义的投影矩阵用于变换场景中的顶点。在发出投影变换命令之前需要调用：

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

投影变换命令描述一种特殊的变换。一般不需要把一个投影变换与另一个投影变换结合起来。

投影变换有 2 种：透视投影和正交投影。投影变换定义视图体，以确定物体如何投影到屏幕上，在视图体外的物体或物体的一部分将被裁剪掉。

6.4.1 透视投影

透视投影(Perspective Projection)的最明显特征是按透视法缩小：物体离照相机越远，成的像就越小。这是由于透视投影的视图体是一个金字塔形的棱台。这种投影方法通常用于动画和视觉模拟，以及其他一些力求真实感的应用。

glFrustum 命令定义一个透视投影矩阵。棱台的四面及其顶和底对应于视图体的 6 个裁剪面(如图 6.11 所示)。在视图体外的物体和物体的一部分被裁剪掉。此外，glFrustum 不要求指定对称的视图体。

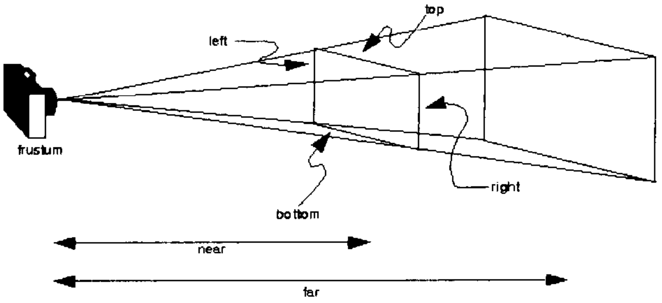

图6.11 利用glFrustum命令指定的透视视图体

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
               GLdouble top, GLdouble near, GLdouble far);
```

用透视矩阵乘当前矩阵。其中：

left，right　　指定左右垂直裁剪面的坐标。

bottom，top　　指定底和顶水平裁剪面的坐标。

near，far　　指定近和远深度裁剪面的距离，两个距离均取正值。

glFrustum() 描述一个产生透视投影的透视矩阵。如果视点定位在原点，那么在近裁剪面上指定的点 (left，bottom，-near)和 (right，top，-near)分别被映射到窗口的左下和右上角。

glFrustum()的概念较容易理解，但使用起来很不直观，利用实用库例程 gluPerspective() 则较为直观。这个例程建立一个与 glFrustum 形状相同的视图体，但用一种不同的方式来指定。gluPerspective()指定 x-y 平面中视域的角度、宽高比和视点到近、远裁剪面之间的距离。这个例程限于沿视线建立 x 和 y 对称的视棱台(如图 6.12 所示)。

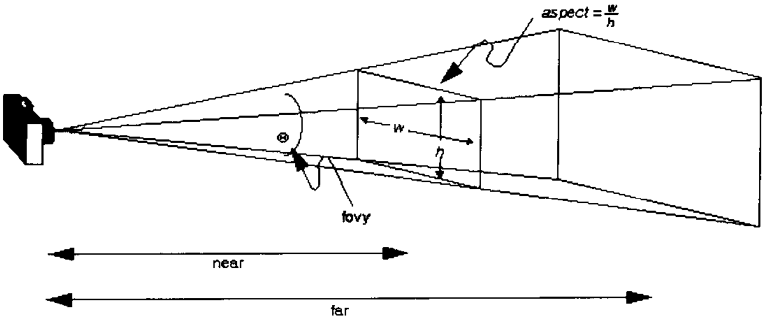

图6.12 利用gluPerspective()指定的透视视图体

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear,
GLdouble zFar);
```

建立透视投影矩阵，并乘当前矩阵。

Fovy　　指定 y 方向中视域的角度(以度为单位)。

Aspect　　指定宽高比，确定 x 方向中的视域。宽高比为 x(宽度)对 y(高度)之比。

ZNear　　指定从观察者(视点)到近裁剪面的距离(该值总是正的)。

ZFar　　指定从观察者(视点)到远裁剪面的距离(该值总是正的)。

gluPerspective()中的宽高比通常应与有关视口相匹配，视域角度在[0.0， 180.0]范围内取值。

6.4.2 正交投影

利用正交投影(Orthographic Projection)，视图体是矩形六面体，或像一个盒子(如图 6.13 所示)。正交投影与透视投影不同，视图体大小从一端到另一端不变化，因此离照相机的距离并不影响场景中物体的大小。这种投影用于生成建筑蓝图和计算机辅助设计等应用。

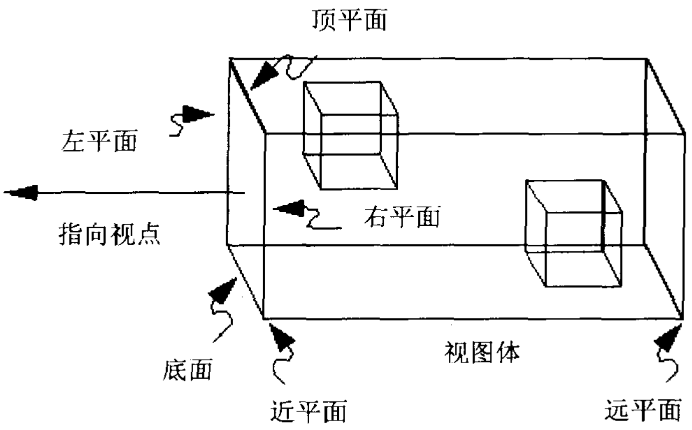

图6.13 正交视图体

glOrtho()命令建立一个正交的平行视图体。glOrtho()指定的参数与 glFrustum()命令相同。

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble
top, GLdouble near, GLdouble far);
```

建立一个正交投影矩阵，并乘以当前矩阵。

left，right	指定左和右垂直裁剪面的坐标。
bottom，top	指定底和顶水平裁剪面的坐标。
near，far	指定视点到近和远深度裁剪面的距离。当这些面在视点的后面时，距离为负值。

glOrtho() 描述一个产生平行投影的透视矩阵。假设视点位于原点，那么(left， bottom，-near)和(right，top，-near)是近裁剪面上指定的点，它们依次映射到窗口的左上和右下角。远裁剪面为在(left，bottom，-far) 和(right，top，-far)上的矩形。当远、近裁剪面在视点之前时，far 和 near 给出 z 负值；当它们出现在视点之后时，给出正值。

在将二维图像投影到二维屏幕上的特殊情况下，可以利用实用库例程 gluOrtho2D()。这个例程与三维形式的 gluOrtho()是一致的，所不同的是，场景中全部物体的坐标假设都位于(-1.0，1.0)之间。

```
gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble
top);
```

建立一个二维正交投影矩阵，并乘当前矩阵。

left，right　　　　指定左和右垂直裁剪面的坐标。

bottom，top　　　　指定底和顶水平裁剪面的坐标。

裁剪面是一个左下角为(left， bottom)，右上角为(right，top)的矩形。

在通过用造型视图矩阵和投影矩阵变换场景中物体的顶点后，任何位于视图体外的顶点都被裁剪掉，定义 6 个裁剪面为视图体的界限。此外，还可以在任意位置指定附加裁剪面。

6.5 视口变换

视口是窗口中的矩形绘图区。图 6.14 表示屏幕中央部分的视口。视口用窗口坐标来度量，它反映了屏幕上的像素位置。视口位于窗口左下角。

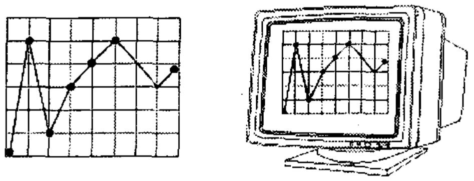

图6.14　视口矩形

6.5.1 定义视口

利用 glViewport 命令选择较小的绘图区，在同一窗口内可以定义多个视口。

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

设置视口，其中：

x，y　　指定视口矩形的左下角(以像素为单位)。默认值为(0， 0)。

width，height　　依次指定窗口的宽和高。

默认情况下，初始视口值为(0， 0， winWidth， winHeight)，其中 winWidth 和 winHeight 为窗口的大小。

视口的宽高比通常等于视图体的宽高比。假如两者的宽高比不同，则投影的图像映射到视口中会发生变形。如图 6.15 所示。

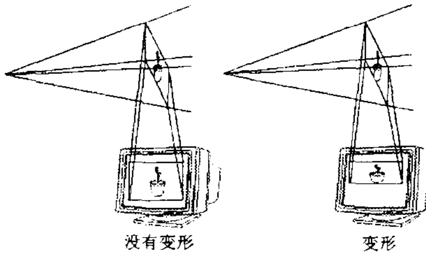

图6.15　将视图体映射到视口

改变窗口大小会影响视口，在应用程序中应检验窗口的缩放事件，并适当修改视口。例如，下面的命令将一个正方形图像映射到一个正方形视口上：

```
gluPerspective(myFovy, 1.0, myNear, myFar);
glViewport(0, 0, 400, 400);
```

而下面的命令将一个不对称的矩形图像投影到一个正方形视口上。如图 6.16 中的右图所示。

```
gluPerspective(myFovy, 2.0, myNear, myFar);
glViewport (0, 0, 400, 400);
```

要避免这种变形，可以利用如下命令：

```
glViewport (0, 0, 400, 200);
```

此外，利用 glViewport 命令可以在同一个窗口中建立多个视口，并可以对每个视口利用不同的投影和视图变换绘制物体。

6.5.2 变换 z 坐标

在经过裁剪和除以 w 之后，z 坐标在-1.0~1.0 之间变化，相应于近和远裁剪面。glDepthRange() 用于指定在这个范围内的单位化 z 坐标线性映射为 z 坐标。不管实际深度缓存是否实现，窗口坐标深度值被处理为在 0.0~1.0 之间变化。

```
void glDepthRange(GLclampd near, GLclampd far);
```

指定从单位化设备坐标到窗口的 z 值映射。

Near　　指定映射到窗口坐标的近裁剪面，默认值为 0。

Far　　指定映射到窗口坐标的远裁剪面，默认值为 1。

near 和 far 值表示可存储在深度缓存中的最小和最大值的配置。默认值依次为 0.0 和 1.0，大多数应用可以利用其默认值。

6.6 操作矩阵栈

前面描述的所有矩阵操作，包括建立、装入和乘以造型视图矩阵，这些操作所形成的矩阵与当前矩阵相乘，成为矩阵栈最上面的项(如图 6.16 所示)。

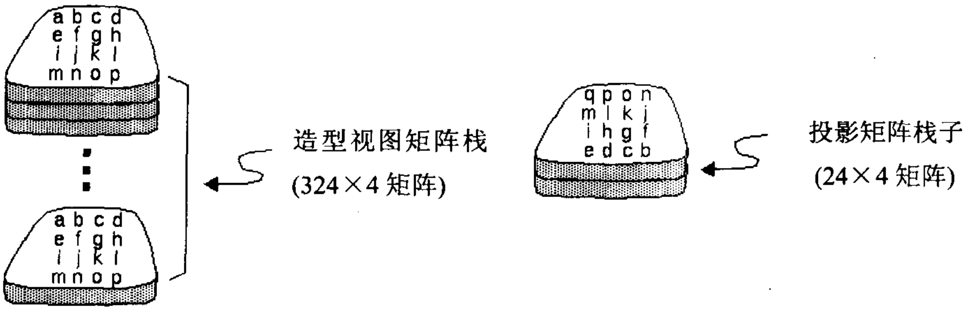

图6.16 造型视图矩阵栈和投影矩阵栈

矩阵栈对构造层次模型是很有用的。通常在层次模型中，复杂物体由简单物体构成。例如一辆汽车有 4 个轮子，每个轮子有 5 个连在车体上的螺栓。利用不同的变换定位轮子，

四次调用绘制轮子的程序就可以绘制四个轮子，同样在绘制每个轮子时，相对于轮子作 5 次不同的变换绘制螺栓，即可绘制轮子上的所有螺栓。因此就必须有一种保存和恢复矩阵的理想机制。

矩阵栈是一种执行矩阵保存和恢复的很理想的机制。利用执行栈操作命令 glPushMatrix() 把当前矩阵保存在栈顶上，而用 glPopMatrix()弹出栈顶上的矩阵，恢复原来的矩阵(如图 6.17 所示)。

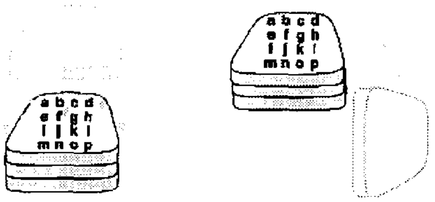

图6.17　推进和弹出矩阵栈

```
void glPushMatrix();
```

在当前栈中把全部矩阵向里推进一层，复制当前矩阵。

```
void glPopMatrix();
```

弹出当前矩阵，用栈上第 2 个矩阵替换当前矩阵。

每一种矩阵方式都各有一个矩阵栈。在 GL_MODELVIEW 方式中，栈深至少为 32。在 GL_PROJECT 和 GL_TEXTURE 方式中，栈深至少为 2。这些矩阵栈根据 OpenGL 的不同实现可能会大于 32 个或 2 个矩阵，利用以下命令可以查到所允许的最大矩阵数：

```
glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, *p);
glGetIntegerv(GL_MAX_PROJECTION_STACK_DEPTH, *p);
glGetIntegerv(GL_MAX_TEXTURE_STACK_DEPTH, *p);
```

下面是推进和弹出矩阵的例子：

```
draw_wheel_and_bolts()
{
  long i;

  draw_wheel();
  for(i=0; i<5; i++){
    glPushMatrix();
      glRotatef(72.0*i, 0.0, 0.0, 1.0);
      glTranslatef(3.0, 0.0, 0.0);
      draw_bolt();
    glPopMatrix();
  }
}

draw_body_and_wheel_and_bolts()
```

```
{
   draw_car_body();
   glPushMatrix();
      glTranslatef(40, 0, 30); /*移到第一个车轮的位置*/
      draw_wheel_and_bolts();
   glPopMatrix();
   glPushMatrix();
      glTranslatef(40, 0, -30); /*移到第二个车轮的位置*/
      draw_wheel_and_bolts();
   glPopMatrix();
   .../*类似地画出最后两个车轮*/
}
```

上面的程序中假设已经存在绘制车体、轮子和螺栓的程序，利用这些程序可以绘制一辆汽车。假设轮子和螺栓与 z 轴是一致的，螺栓离轮子中心 3 个单位，并分别均匀相隔 72°，前轮在汽车原点前面 40 个单位和在汽车原点右边和左边 30 个单位。利用栈比利用单独的矩阵有效，尤其当用硬件实现栈时效率更高。

6.7　附加裁剪面

除了视图体的 6 个裁剪面(左、右、底、顶、近和远)外，还可以定义最多 6 个附加裁剪面来进一步限制视图体，如图 6.18 所示.

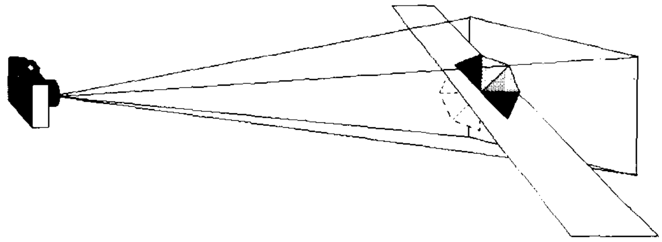

图6.18　附加裁剪面和视图体

附加裁剪面可用于显示物体的剖面图等情况下。每个裁剪面是通过指定方程 Ax+By+cz+D=0 中的系数来确定的。裁剪面通过造型和视图变换自动进行相应的变换。最后的裁剪体成为视图体和附加裁剪面所定义的全部半空间的交。

```
void glClipPlane(GLenum plane, const GLdouble *equation);
```

定义附加裁剪面，其中：

plane　用符号名 GL_CLIP_PLANEi 指定裁剪面，其中 i 为 0~5 之间的一个整数，指定 6 个裁剪面中的一个。

equation　指定 4 个值的数组，存放平面方程的 4 个系数。

满足 $(ABCD)M^{-1}(x_e, y_e, z_e, w_e)^T>=0$ 的全部人眼坐标 (x_e, y_e, z_e, w_e) 的点位于由这个

平面定义的半空间中，其中 M 为调用 glClipPlane()时的当前造型视图矩阵。

在定义每个附加裁剪面之前，必须发出激活附加裁剪面命令：

```
glEnable(GL_CLIP_PLANEi);
```

用以下命令可以关闭一个附加裁剪面：

```
glDisable(GL_CLIP_PLANEi);
```

有的 OpenGL 实现允许多于 6 个裁剪面，可以用下面的命令查询支持裁剪面的数目:

```
glGetIntegerv(GL_MAX_CLIP_PLANES, *p);
```

利用裁剪面的程序示例如下，本示例绘制一个线框球体，用 2 个裁剪面切去原始球体的 3/4，如图 6.19 所示。

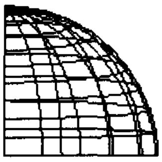

图6. 19　裁剪后的线框球体

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void display(void)
{
   GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};     /* y < 0 */
   GLdouble eqn2[4] = {1.0, 0.0, 0.0, 0.0};    /* x < 0 */

   glClear(GL_COLOR_BUFFER_BIT);

   glColor3f (1.0, 1.0, 1.0);
   glPushMatrix();
   glTranslatef (0.0, 0.0, -5.0);

   glClipPlane (GL_CLIP_PLANE0, eqn);
   glEnable (GL_CLIP_PLANE0);
   glClipPlane (GL_CLIP_PLANE1, eqn2);
   glEnable (GL_CLIP_PLANE1);
   glRotatef (90.0, 1.0, 0.0, 0.0);
   auxWireSphere(1.0);
   glPopMatrix();
   glFlush();
```

```
}
void myinit (void)
{
    glShadeModel (GL_FLAT);
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

6.8 利用几种变换综合绘制机器人手臂

本节说明多节机器人手臂的绘制程序。手臂应利用肩、肘或其他结合点上的支点相连接。图 6.20 表示仅有单个结合点的手臂。

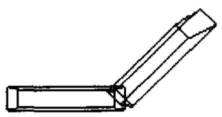

图6.20 机器人手臂

绘制机器人手臂的各节，首先需要用适当的造型变换来定向每个节。由于局部坐标系统的原点最初在节的中心，所以需要将局部坐标系统移到节的一端上，否则将绕节的中心而不是绕支点旋转。

利用 glTranslate*()建立支点，然后利用 glRotate*()在节的枢纽上转动，再移回到节的

中心，最后再绘制节。下面是绘制手臂的程序：

```
#include <math.h>
#include <GL/glaux.h>
#include <GL/gl.h>
#include <GL/glu.h>
#define HINGE 1
#define ARM 2
#define HAND 3
void CALLBACK CameraLeft(GLvoid);
void CALLBACK CameraRight(GLvoid);
void CALLBACK CameraForward(GLvoid);
void CALLBACK CameraBack(GLvoid);
void BuildHinge();
void BuildArm();
void BuildHand();
void BuildBox (GLfloat xdim, GLfloat ydim, GLfloat zdim);
void DrawBase();
void DrawVertical_Shaff();
void DrawHinge();
void DrawArm();
void DrawHorizontal_Shaff();
void DrawHand();
void Reset();
// 照相机变量
void MoveCamera(GLdouble dStep);
//  将弧度转换为角度
#define dRadToDeg(x) (((float)x)*57.29577951308f)
#define fsin(x) (float)sin(x)
#define fcos(x) (float)cos(x)
#define glRGB(x, y, z) glColor3ub((GLubyte)x, (GLubyte)y, (GLubyte)z)

GLfloat m_camera_radsFromEast;
GLfloat m_Translate_x = 0.0f;
GLfloat m_Translate_z = 30.0f;
static GLdouble PI = 3.14159265359;
GLfloat light0Position[]={0.0f, 5.0f, 0.0f, 1.0f};
GLfloat m_cameraTranslationX = 0.0f;
GLfloat m_cameraTranslationY = 0.0f;
GLfloat m_cameraTranslationZ = 50.0f;
```

```
GLfloat m_translationUnit = 1.0f;   //  每按一次键的移动单位

GLint   m_view = 0;
GLint   m_uview = 0;

GLfloat m_xPos=0.0f;
GLfloat m_zPos=-30.0f;
GLfloat m_yaw=90.0f;

// 机器人大小
GLfloat m_scale_x=0.1f;
GLfloat m_scale_y=0.1f;
GLfloat m_scale_z=0.1f;
GLfloat m_divisor=1.0f;
GLfloat m_robot_radsFromEast;
// 旋转控制变量
GLint m_Vertical_Shaff_Angle=0;
GLint m_Hinge_Angle=0;
GLint m_Horizontal_Shaff_Angle=0;

void BuildBox (GLfloat xdim, GLfloat ydim, GLfloat zdim) {

  glBegin(GL_POLYGON);  /* 前平面*/
    glNormal3f(0.0f, 0.0f, 1.0f);
    glVertex3f(xdim/2.0f, ydim/2.0f, zdim/2.0f);
    glVertex3f(xdim/2.0f, -ydim/2.0f, zdim/2.0f);
    glVertex3f(-xdim/2.0f, -ydim/2.0f, zdim/2.0f);
    glVertex3f(-xdim/2.0f, ydim/2.0f, zdim/2.0f);
  glEnd();

  glBegin(GL_POLYGON);  /*后平面*/
    glNormal3f(0.0f, 0.0f, -1.0f);
    glVertex3f(xdim/2.0f, ydim/2.0f, -zdim/2.0f);
    glVertex3f(xdim/2.0f, -ydim/2.0f, -zdim/2.0f);
    glVertex3f(-xdim/2.0f, -ydim/2.0f, -zdim/2.0f);
    glVertex3f(-xdim/2.0f, ydim/2.0f, -zdim/2.0f);
  glEnd();
```

```
    glBegin(GL_POLYGON); /*右面*/
      glNormal3f(1.0f, 0.0f, 0.0f);
      glVertex3f(xdim/2.0f, ydim/2.0f, zdim/2.0f);
      glVertex3f(xdim/2.0f, ydim/2.0f, -zdim/2.0f);
      glVertex3f(xdim/2.0f, -ydim/2.0f, -zdim/2.0f);
      glVertex3f(xdim/2.0f, -ydim/2.0f, zdim/2.0f);
    glEnd();

    glBegin(GL_POLYGON); /*右平面*/
      glNormal3f(-1.0f, 0.0f, 0.0f);
      glVertex3f(-xdim/2.0f, ydim/2.0f, zdim/2.0f);
      glVertex3f(-xdim/2.0f, ydim/2.0f, -zdim/2.0f);
      glVertex3f(-xdim/2.0f, -ydim/2.0f, -zdim/2.0f);
      glVertex3f(-xdim/2.0f, -ydim/2.0f, zdim/2.0f);
    glEnd();

    glBegin(GL_POLYGON); /*顶平面*/
      glNormal3f(0.0f, 1.0f, 0.0f);
      glVertex3f(xdim/2.0f, ydim/2.0f, zdim/2.0f);
      glVertex3f(xdim/2.0f, ydim/2.0f, -zdim/2.0f);
      glVertex3f(-xdim/2.0f, ydim/2.0f, -zdim/2.0f);
      glVertex3f(-xdim/2.0f, ydim/2.0f, zdim/2.0f);
    glEnd();

    glBegin(GL_POLYGON); /*底平面*/
      glNormal3f(0.0f, -1.0f, 0.0f);
      glVertex3f(xdim/2.0f, -ydim/2.0f, zdim/2.0f);
      glVertex3f(xdim/2.0f, -ydim/2.0f, -zdim/2.0f);
      glVertex3f(-xdim/2.0f, -ydim/2.0f, -zdim/2.0f);
      glVertex3f(-xdim/2.0f, -ydim/2.0f, zdim/2.0f);
    glEnd();
}
void Reset()
{
    double  PI = 3.14159265359;
    m_view = 0;
    m_uview = 0;
    m_xPos=0.0f;
    m_zPos=0.0f;
```

```
        m_camera_radsFromEast=(GLfloat)PI/2.0f;
        m_yaw=90.0f;
        // 机器人大小
        m_scale_x=0.9f;
        m_scale_y=0.9f;
        m_scale_z=0.9f;

        m_translationUnit=5.0f;
}
GLvoid CALLBACK Vertical_Shaff_Add (GLvoid)
{
    m_Vertical_Shaff_Angle = (m_Vertical_Shaff_Angle + 2) % 360;
}

GLvoid CALLBACK Vertical_Shaff_Subtract (GLvoid)
{
    m_Vertical_Shaff_Angle = (m_Vertical_Shaff_Angle -2) % 360;
}

GLvoid CALLBACK Hinge_Add (GLvoid)
{
     if (m_Hinge_Angle<25)
     {
         m_Hinge_Angle = (m_Hinge_Angle + 1);
     }
}
GLvoid CALLBACK Hinge_Subtract (GLvoid)
{
     if (m_Hinge_Angle>-25)
     {
         m_Hinge_Angle = (m_Hinge_Angle -1);
     }
}
GLvoid CALLBACK Horizontal_Shaff_Add (GLvoid)
{
   m_Horizontal_Shaff_Angle = (m_Horizontal_Shaff_Angle + 10) % 360;
}

GLvoid CALLBACK Horizontal_Shaff_Subtract (GLvoid)
```

```
{
   m_Horizontal_Shaff_Angle = (m_Horizontal_Shaff_Angle -10) % 360;
}
void ViewSubtract()
{
   m_view = (m_view -15) % 360;
}

void UviewAdd()
{
    m_uview = (m_uview + 15) % 360;
}

void UviewSubtract()
{
    m_uview = (m_uview -15) % 360;
}
// 绘制机器人底座
void DrawBase (void)
{
    GLUquadricObj *obj = gluNewQuadric();
    gluQuadricDrawStyle(obj, GLU_FILL);
    gluQuadricNormals(obj, GLU_SMOOTH);
    gluQuadricOrientation(obj, GLU_OUTSIDE);
     glPushMatrix();        // 底座圆柱
       glTranslatef(0.0f, 5.0f, 0.0f);   // 绕X 轴旋转
      glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
        gluCylinder(obj, 3.0f, 4.0f, 10.0f, 100, 1);
     glPopMatrix();
    glPushMatrix();
      glTranslatef(0.0f, 5.0f, 0.0f);
      glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
      glTranslatef(0.0f, 0.0f, 10.0f);
     gluDisk(obj, 0.0f, 3.0f, 10, 10);
    glPopMatrix();
     DrawVertical_Shaff();
    gluDeleteQuadric(obj);
}
void DrawVertical_Shaff (void)
```

```
{
    GLUquadricObj *obj = gluNewQuadric();
    gluQuadricDrawStyle(obj, GLU_FILL);
    gluQuadricNormals(obj, GLU_SMOOTH);
    gluQuadricOrientation(obj, GLU_OUTSIDE);
    glPushMatrix();
      glRotatef((GLfloat)m_Vertical_Shaff_Angle, 0.0f, 1.0f, 0.0f);
      glTranslatef(0.0f, 10.0f, 0.0f);
      glPushMatrix();
           glTranslatef(0.0f, 5.0f, 0.0f);
           glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
           gluCylinder(obj, 1.0f, 1.0f, 10.0f, 100, 1);
       glPopMatrix();
       glPushMatrix();
           glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
           glTranslatef(0.0f, 0.0f, 10.0f);
           gluDisk(obj, 0.0f, 1.0f, 10, 10);
           /*画绞链*/
       glPopMatrix();
    DrawHinge();
   glPopMatrix();
  gluDeleteQuadric(obj);
}
void DrawHorizontal_Shaff(void)
{
    GLUquadricObj *obj = gluNewQuadric();
    gluQuadricDrawStyle(obj, GLU_FILL);
    gluQuadricNormals(obj, GLU_SMOOTH);
    gluQuadricOrientation(obj, GLU_OUTSIDE);
    glPushMatrix();
        glRotatef((GLfloat)m_Horizontal_Shaff_Angle, 1.0f,0.0f,0.0f);

        glPushMatrix();
        glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
        gluCylinder(obj, 0.5f, 0.5f, 10.0f, 100, 1);
        glPopMatrix();
        glPushMatrix();
            glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
            glTranslatef(0.0f, 0.0f, 10.0f);
```

```
            gluDisk(obj, 0.0f, 0.5f, 100, 10);
        glPopMatrix();
        DrawHand();
    glPopMatrix();
}
void DrawHinge(void)
{
    // 相对于父结点的支点
    /*Hinge_Add(); */
    glPushMatrix();
        glTranslatef(0.0f, 5.0f, 0.0f);
        glRotatef((GLfloat)m_Hinge_Angle, 0.0f, 0.0f, 1.0f);
        glCallList(HINGE);
        DrawArm();
    glPopMatrix();
}
void DrawArm(void)
{
    glCallList(ARM);
    glTranslatef(5.0f, 0.0f, 0.0f);
    DrawHorizontal_Shaff();
}
void DrawHand(void)
{
    glTranslatef(10.0f, 0.0f, 0.0f);
    glCallList(HAND);
}
void BuildHinge(void)
{
    glPushMatrix();
    glNewList(HINGE, GL_COMPILE);
        BuildBox(4.0f, 1.0f, 4.0f);   // 底
        glTranslatef(0.0f, 2.50f, 2.5f);
        BuildBox(4.0f, 6.0f, 1.0f);    // 一边
        glTranslatef(0.0f, -2.50f, -2.5f);
        glTranslatef(0.0f, 2.50f, -2.5f);
        BuildBox(4.0f, 6.0f, 1.0f); // 另一边
        glTranslatef(0.0f, -2.50f, 2.5f);
    glEndList();
```

```
    glPopMatrix();
}
void BuildHand(void)
{
    glPushMatrix();
    glNewList(HAND, GL_COMPILE);
        glRotatef(-90.0f, 0.0f, 0.0f, 1.0f);
        BuildBox(4.0f, 1.0f, 4.0f);     // 底
        glTranslatef(0.0f, 2.50f, 2.5f);
        BuildBox(4.0f, 6.0f, 1.0f);     // 一边
        glTranslatef(0.0f, -2.50f, -2.5f);
        glTranslatef(0.0f, 2.50f, -2.5f);
        BuildBox(4.0f, 6.0f, 1.0f);    // 另一边
        glTranslatef(0.0f, -2.50f, 2.5f);
    glEndList();
    glPopMatrix();
}
void BuildArm (void)
{
    glPushMatrix();
    glNewList(ARM, GL_COMPILE);
    glTranslatef(8.0f, 2.25f, 0.0f);
    BuildBox(20.0f, 3.5f, 3.5f);
    glEndList();
    glPopMatrix();
}
void DrawWielder()
{
   double     PI =  3.14159265359;
 // 开始绘制整个模型
 glPushMatrix();
    glColor4f (1.0f,  1.0f,  1.0f,  0.2f);
   glScalef(m_scale_x, m_scale_y, m_scale_z);  // 设置机器人的大小
   glTranslatef(m_xPos, 0.0f, m_zPos);        // 旋转和平移机器人的位置
   glRotatef(dRadToDeg(m_robot_radsFromEast+((float)PI/2.0f))-90
0.0f,  1.0f,  0.0f);
    DrawBase();
 glPopMatrix();
}
```

```
void CreateWielder()
{
  // 初始化显示列表
  BuildHinge();
  BuildArm();
  BuildHand();
  Reset();
}
void CALLBACK Idle_Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(m_Translate_x , m_cameraTranslationY , m_Translate_z ,
    m_Translate_x+(100.0f*(GLfloat)cos(m_camera_radsFromEast)) , m_
    cameraTranslationY, m_Translate_z-(100.0f*(GLfloat)sin (m_camera_
    radsFromEast)), 0.0f, 1.0f, 0.0f);
    DrawWielder();
    glFlush();
    auxSwapBuffers();

}
void CALLBACK Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(m_Translate_x, m_cameraTranslationY, m_Translate_z,
m_Translate_x+(100.0f*(GLfloat)cos(m_camera_radsFromEast)), m_camera-
Translation Y, m_Translate_z - (100.0f*(GLfloat)sin (m_camera_rads-
FromEast)), 0.0f, 1.0f, 0.0f);
    DrawWielder();
    glFlush();
    auxSwapBuffers();
}

void CALLBACK myinit (void)
{
        // 指定物体的初始材质、光源和位置
    GLfloat mat_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat mat_shininess[] = {100.0f};
```

```
    GLfloat light_ambient[] = {0.0f, 0.0f, 0.0f, 1.0f};
    GLfloat light_diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_position[] = {0.0f, 100.0f, 150.0f, 0.0f};

    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);

    glShadeModel(GL_SMOOTH);

    // 背景色
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    // 原始角
    m_camera_radsFromEast=(GLfloat) (2.0f*PI)/4.0f;
    // 创建机器人
    CreateWielder();
}

void CALLBACK MyReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(65.0f, (GLfloat) w/(GLfloat) h, 1.0f, 140.0f);
    glMatrixMode(GL_MODELVIEW);
}
void MoveCamera(GLdouble dStep)
{
    GLdouble xChange, zChange;
```

```
        xChange = dStep*cos(m_camera_radsFromEast);
        zChange = -dStep*sin(m_camera_radsFromEast);
        // 平移变量增加
        m_Translate_x += (GLfloat)xChange;
        m_Translate_z += (GLfloat)zChange;
    }
    void CALLBACK CameraLeft(void)
    {
            m_camera_radsFromEast +=(GLfloat) PI/30.0f;
            if(m_camera_radsFromEast > (2.0*PI))          // 防止越界
                m_camera_radsFromEast = 0.0f;
    }

    GLvoid CALLBACK CameraRight(GLvoid)
    {
        m_camera_radsFromEast-= (GLfloat) PI/30.0f;
        if (m_camera_radsFromEast <0.0)       // 防止越界
          m_camera_radsFromEast = (GLfloat)(2.0*PI);
    }
    GLvoid CALLBACK CameraForward(GLvoid)
    {
        MoveCamera((double)m_translationUnit);
    }
    GLvoid CALLBACK CameraBack(GLvoid)
    {
        MoveCamera((GLdouble)(-1.0f*m_translationUnit));
    }
    //  主循环，打开窗口并初始化窗口大小、标题
    //  设置为 RGBA 显示模式，指定如何处理输入事件
    int main(int argc, char** argv)
    {
        auxInitDisplayMode (AUX_DOUBLE | AUX_RGBA | AUX_DEPTH);
        auxInitPosition (0, 0, 600, 550);
        auxInitWindow ("Up, down, left, right Vertical Staff 1&2, Hinge 3&4,
    Hand 5&6");
        myinit ();
        auxKeyFunc (AUX_LEFT, CameraLeft);
        auxKeyFunc (AUX_RIGHT, CameraRight);
```

```
        auxKeyFunc (AUX_UP, CameraForward);
        auxKeyFunc (AUX_DOWN, CameraBack);
        auxKeyFunc (AUX_1, Vertical_Shaff_Add);
        auxKeyFunc (AUX_2, Vertical_Shaff_Subtract);
        auxKeyFunc (AUX_3, Hinge_Add);
        auxKeyFunc (AUX_4, Hinge_Subtract);
        auxKeyFunc (AUX_5, Horizontal_Shaff_Add);
        auxKeyFunc (AUX_6, Horizontal_Shaff_Subtract);
        auxReshapeFunc (MyReshape);
        auxIdleFunc(Idle_Display);
        auxMainLoop(Display);
        return 0;
    }
```

程序运行结果如图 6.21 所示。

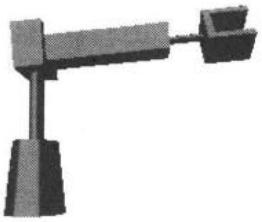

图6. 21 机器人手臂程序运行结果

第 7 章　OpenGL 颜色

要生成高度真实感的图形，就必须对颜色进行讨论。颜色是一门非常复杂的学科，涉及到数学、物理学、心理学和美学等领域。这里仅讨论与计算机图形学有关的部分。

利用计算机在图形设备上生成连续色调的真实感图形必须完成以下 4 项基本任务：

- 建模

用数学方法建立所需三维场景的几何描述，并将它们输入计算机。这部分工作可由三维立体造型或曲面造型系统来完成。场景的几何描述将直接影响图形的复杂性和图形绘制的计算时间耗费，选择合理有效的数据表示和输入手段是极其重要的。

- 透视

将三维几何描述转换为二维透视图，这可以通过对场景的透视变换来完成。

- 可见面判定

确定场景中的所有可见面，这需要使用隐藏面消除算法，将可视域之外的不可见面消去。

- 计算场景中可见面的颜色

根据基于光学物理的光照明模型，计算可见面投射到观察者眼中的光亮度大小和色彩组成，并将它转换成适合图形设备的颜色值，从而确定投影画面上每一像素的颜色，最终形成图形。

由此可见，计算和设置场景中的颜色对于绘制高度真实感的场景是非常重要的。它是形成最终结果的关键一步。若掌握好颜色的运用，就能走进缤纷绚丽的色彩世界，从中享受无穷的乐趣。

7.1　色彩视觉

颜色是可见光的一种视觉特性。从客观上讲，可见光本身并没有颜色，但可见光不同的光谱分布刺激人眼使之产生颜色的感觉，因此，颜色是一种主观评判的概念。可见光的波长范围大约是 380~780nm，即只有从 380~780nm 波长范围内的电磁辐射才能够引起人的感觉。

物体的颜色不仅取决于物体本身，它还与光源、周围环境的颜色，以及观察者的视觉系统都有关系。例如，有些物体(如粉笔、纸张)只反射光线，而另外有些物体(如玻璃、水)不仅会反射光，而且会透射光。当一个只反射纯绿色的表面用纯红色光照明时，呈黑色。类似地，从一块只透蓝光的玻璃后面观察一道红光，玻璃也是呈黑色。

从视觉的角度，颜色包含 3 个要素：色彩(hue)、饱和度(saturation)和亮度(lightness)。所谓色彩，就是通常所说的红绿蓝紫等，是使一种颜色区别于另一种颜色的要素。饱和度

就是颜色的纯度。在某种颜色中添加白色相当于减少该颜色的饱和度。例如，鲜红色的饱和度高，粉红色的饱和度低。亮度，即光的强度。

为了更客观地、定量地阐明颜色的概念，引进光学的几个术语：主波长(domination wavelength)、纯度(purity)和辉度(luminance)。主波长是观察光线所见的光的波长，对应于视觉所感知的色彩。光的纯度对应于颜色的饱和度。辉度则是颜色的亮度。一种颜色光的纯度是定义该颜色光的(主波长的)纯色光与白色光的比例。每一种纯色光都是百分之百饱和的，因而不包含任何白色光。

从根本上讲，光是波长为 380~780nm 的电磁波。这些电磁波被视觉系统感知为紫、青、蓝、绿、黄、橙、红等颜色。事实上可以用主波长、纯度和辉度三元组来简明地描述任何光谱分布的视觉效果。许多不同光谱分布产生的视觉效果(颜色)是一样的。换句话说，光谱与颜色的对应是多对一的。光谱分布不同而看上去颜色相同称为条件等色。

彩色显示器(CRT)上每个像素都是由红、绿、蓝 3 种荧光点组成。这是以人类视觉感知的三刺激理论为基础设计的。三刺激理论基于这样一个假设：人眼的视网膜中有 3 种锥状视觉细胞，分别对红、绿、蓝 3 种光最敏感。

大量实验表明，人眼大约可分辨 35 万种颜色，但只能分辨 128 种不同的色彩。在可见光谱的两端附近，人眼可以区别出波长相差 10nm 左右的不同色彩。而在光谱的蓝色至黄色区间，却可以分辨出波长仅相差 1nm 左右的 2 种不同色彩。人眼分辨饱和度的能力比分辨色彩的能力差，对红、紫色只能分辨出 23 种不同的饱和度，而对于黄色，仅能分辨出 16 种不同的饱和度。

7.2 计算机颜色

7.2.1 颜色生成原理

在一台计算机彩色显示屏上，硬件(即监视器内部的电子枪)使屏幕上的每个像素发出不同量的红、绿、蓝光，称为 R、G、B 值。这些值通常被组装在一起(有时加入第 4 个值，称为 alpha)，合起来称为 RGB(RGBA)值。每个像素的颜色信息，可以用 RGB(或 RGBA)方式存储，也可以用颜色索引方式存储。颜色索引方式指每个像素的颜色信息存储单个值(称为颜色索引)。每个颜色索引定义一组 R、G、B 值，作为表中的一项。这种表称为查色表，表中的值可以改变。

不同的图形硬件所具有的像素阵列大小以及每个像素上可显示的颜色数目也有很大不同。在一给定的图形系统中，每个像素具有相同数量的存储区，所有像素的全部存储区称为显示缓存。缓存大小通常以字节来度量，通常所说的带2MB缓存的显示卡就是指显示卡中的缓存大小为2MB字节。缓存的大小因机器中显示卡的不同而不同。对于用作图形处理的机器而言，应尽可能选用有较大显示缓存和较快刷新速度的显示卡。有些显示卡还专门针对3D图形处理作了加速。利用这样的显示卡可以更好地发挥三维图形的各种优良性能。

7.2.2 RGB 颜色模型

颜色模型指的是某个三维颜色空间中的一个可见光子集。它包含某个颜色域的所有颜色。例如，RGB 颜色模型是三维直角坐标颜色系统中的一个单位立方体。它的用途是在某个颜色域内方便地指定颜色。由前面知道，任何颜色域都只是可见光的一个子集，所以任何一个颜色模型都无法包含所有的可见光。

大多数彩色图形显示器使用红、绿、蓝三原色，因而 RGB 颜色模型是人们最熟悉，也是最简单的一种颜色模型。RGB 颜色模型采用直角坐标系。红、绿、蓝原色是加性原色，也就是说，各个原色的光叠加在一起能产生复合色，如图 7.1 所示。

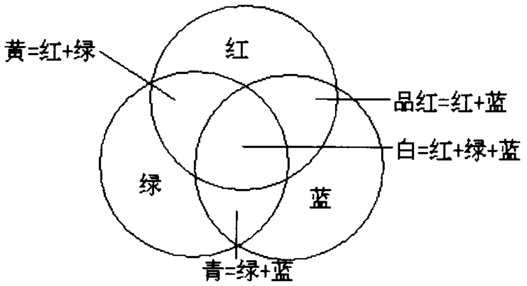

图7.1 RGB三原色叠加效果示意图

RGB 颜色模型通常用如图 7.2 所示的单位立方体来表示。在立方体的对角线上，各原色的量相等，产生由暗到亮的白色，即灰度。(0.0, 0.0, 0.0)为黑，(1.0, 1.0, 1.0)为白。正方体的其他 6 个角点分别为红、黄、绿、蓝、青和品红。

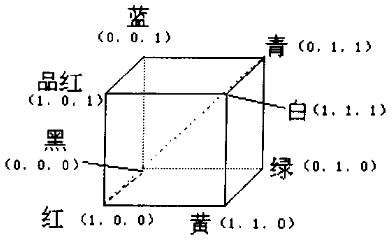

图7.2 RGB立方体

7.3 颜色模式

在 OpenGL 中通常使用 2 种颜色模式，即 RGBA 模式和颜色索引模式。在颜色索引模式下，对每个像素存储一定数量的颜色数据。这个量是由帧缓存中的位平面数所确定的。如果有 8 个颜色位平面，则每个像素有 8 个颜色位，从而对像素可以存储 2^8=256 种不同的值或颜色。现在的图形卡硬件环境，一般都支持 24 位颜色位面，通常对 R，G，B，A 都分配 8 位，但并非一定得如此。可以调用 glGetIntegerv()函数，将参数分别设置为：GL_RED_BITS ， GL_GREEN_BITS ， GL_BLUE_BITS ， GL_ALPHA_BITS ，

GL_INDEX_BITS，来查询有关 R，G，B，A 或颜色索引值的位面数。

7.3.1 RGBA 颜色模式

在 RGBA 模式下，硬件对 R，G，B，A 分量每个配置一定数目的位面(不同分量位面数不一定相同)。在颜色位面中，R、G、B 的值一般是用整数存储，而不是浮点数。但为了存储和检索的方便，还是把这些整数换算成 0.0~1.0 之间的浮点数。一个 8 位颜色分量，可以表示成 0~255 之间的整数，因此换算到 0.0~1.0 之间的浮点数的公式为：i/255，其中 i 为该整数。

单个像素可以显示的不同颜色的数目，取决于位面的数目和硬件解释位面的能力。不同颜色的数目不能超过 2^n 种，n 是位面数。因此，一台 RGB 为 24 位的计算机最多能显示 16.77M 种不同的颜色。

7.3.2 颜色索引模式

在颜色索引模式下，OpenGL 利用一个查色表，这有些类似于画家用来调制颜色的调色板，用准备好编号的颜料来绘图。同样，计算机的查色表，是用索引来表示各种原始的红、绿、蓝的混合。

就加画家从调色板中选择各种颜色绘图，计算机对每个像素的颜色索引存储在位面中，这些位面值引用查色表，用查色表中相应的红、绿、蓝值在屏幕上绘图。

在颜色索引模式下，同时可用的颜色数目受到查色表大小和可用位面数的限制。查色表大小是硬件确定的，通常在 256~4096 之间。查色表的大小为 2 的幂次，可通过颜色索引模式下的位面数来检索。如果在查色表中有 2^n 个索引和 m 个可用的位面，那么可用的颜色数目为 2^n 和 2^m 中较小的。

注意：*在 RGBA 模式下，每个像素的颜色与其他像素是无关的。但是在颜色索引模式下，存储在位面中的每个具有相同索引的像素共用查色表的相同位置。假如改变查色表中某个索引的内容，全部具有这种颜色的像素都被改变。*

7.3.3 两种模式的应用场合

使用 RGBA 模式还是使用颜色索引模式要根据所用的硬件和应用的需求而定。对于大多数系统，使用 RGBA 模式可同时表示的颜色较使用颜色索引模式可同时表示的颜色要多。另外，对于明暗处理、光照、纹理映射和雾化等特殊效果，使用 RGBA 模式较使用颜色索引模式更灵活。

但在以下情况下，可考虑使用颜色索引模式：

- 当移植一个原来主要用颜色索引模式的应用时，最好不去改变原来的颜色模式。
- 当可用的位面只有很少的数目，而且只需使用少于 2^n 种不同的颜色时。
- 当只有有限数目的位面可用时，RGBA 模式会产生很不精确的明暗处理。如只要求给出灰度之类的有限度的明暗处理时，颜色索引方式则可以处理得较好。

- 颜色索引模式有一些技巧性用途，例如，查色表动画等，在后面将给出这样一个例子来说明颜色索引模式其实也可以绘制很漂亮的图形。

7.4 指定颜色和明暗处理模型

7.4.1 在 RGBA 模式下指定颜色

在 RGBA 模式下，利用 glColor*命令选取当前颜色。glColor*命令有如下几种形式：

```
void glColor3{ b s i f d ub us ui }(r, g, b:TYPE);
void glColor4{ b s i f d ub us ui }(r, g, b:TYPE; a:TYPE);
void glColor3{ b s i f d ub us ui }v(r, g, b:TYPE);
void glColor4{ b s i f d ub us ui }v(r, g, b:TYPE; a:TYPE);
```

其中，参数 a 是表示透明度的 alpha 值。后面两个带 v 后缀的命令表明他们的参数是向量，即数组。以 glColor3f 为例，其参数取值范围是[-1.0，1.0]，这是帧缓存中允许的最小值和最大值，当参数值不在该范围内时，将自动把它的值强置于[-1.0，1.0]之间。其他数值类型的函数也将自动把参数均匀映射到这个区间，映射范围如表 7.1 所示。

表7.1 函数的映射范围

后 缀	类 型	最 小 值	映射最小值	最 大 值	映射最大值
B	单字节整型	-128	-1.0	127	1.0
s	双字节整型	-32 768	-1.0	32 767	1.0
i	四字节整型	-2 147 483 648	-1.0	2 147 483 647	1.0
ub	无符号单字节整型	0	0.0	255	1.0
us	无符号双字节整型	0	0.0	65 535	1.0
ui	无符号四字节整型	0	0.0	4 294 967 295	1.0

不同数值类型的参数映射到[-1.0，1.0]区间遵循的转换公式如表 7.2 所示。

表7.2 不同数据类型参数映射转换公式

数值类型	转换公式
ub	$c/(2^8-1)$
b	$(2c+1)/(2^8-1)$
us	$c/(2^{16}-1)$
s	$(2c+1)/(2^{16}-1)$
ui	$c/(2^{32}-1)$
i	$(2c+1)/(2^{32}-1)$
f	c
d	c

7.4.2　在颜色索引模式下指定颜色

OpenGL 颜色模式一共有两种：RGBA 模式和颜色索引模式。在 RGBA 模式下，所有的颜色定义全用 R，G，B3 个值来表示，有时也加上 alpha 值，即 RGBA 模式。在颜色索引模式下，每一个像素的颜色是用颜色索引表中的某个颜色索引表示，而这个索引值指向了相应的 R、G、B 值。这样的颜色索引表称为颜色映射(Color Map)。

一般来说，RGBA 模式适用于纹理映射，以及较好地应用于光照、明暗处理、雾、反走样和混和等。

在颜色索引模式下，指定颜色可以使用如下两条命令：

```
void glIndex{s i f d}(c:TYPE);
void glIndex{s i f d}(c:PTYPE);
```

这 2 条命令将从颜色索引表中选取颜色索引值。参数 c 是待设置的当前颜色索引值(调色板号)，当 c 的取值大于颜色索引总数时，对 c 取模。7.5“颜色索引模式下指定颜色实例过程”一节给出一个在颜色索引模式下指定颜色的例子，演示简单颜色索引的应用。程序中颜色索引方式绘制一个用自定义的三角形带表示的圆环，并通过对它进行变换形成一个摇摆式的动画。

7.5　颜色索引模式下指定颜色实例过程

下面先简单介绍颜色索引模式下指定颜色的基本方法，然后通过一个具体而生动的实例来讲解这种方法。

7.5.1　在颜色索引模式下指定颜色

在颜色索引模式下，OpenGL 利用一个查色表，就象画家利用混合颜料的调色板，用准备好编号的颜料来绘图。同样，计算机的查色表，是用索引表示各种原始红、绿、蓝色的混合，如图 7.3 所示。

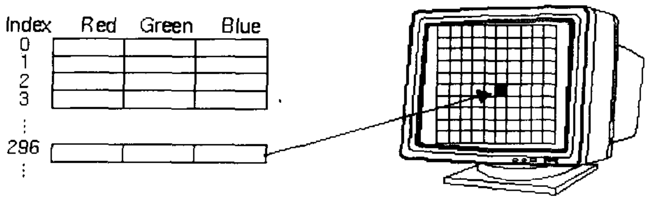

图7.3　查色表

类似于画家从调色板中选择各种颜色绘图，计算机对每个像素的颜色索引存储在位面中，这些位面值引用查色表，用查色表中相应的红、绿、蓝值在屏幕上绘图，如图 7.4 所示。

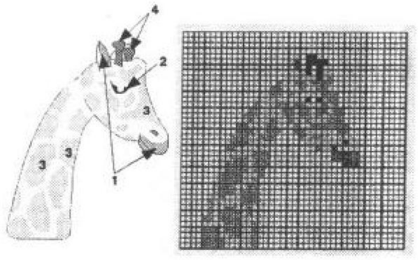

图7.4　利用查色表绘图

在颜色索引模式下，同时可用的颜色数目受到查色表大小和可用位面数的限制。查色表大小是硬件专用于查色表的总数来确定的，通常在 256 到 4096 之间。查色表为 2 的幂次，通过可用于颜色索引模式下的位面数来检索。如果在查色表中有 2^n 个索引和 m 个可用的位面，那么可用的颜色数目为 2^n 和 2^m 中较小的。

在 RGBA 模式下，每个像素的颜色与其他像素是无关的。但是在颜色索引模式下，存储在位面中的每个具有相同索引的像素共用查色表的相同位置。假如改变查色表中某个索引的内容，全部具有这种颜色的像素都被改变。

在颜色索引模式下，利用 glIndex*()命令选取一个单值的颜色作为当前的颜色索引。可以调用 glIndex{sifd}(TYPE c)或其矢量形式：glIndex{sifd}v(Const TYPE c)来设置当前颜色索引。其中，参数 c 用于指定当前颜色索引的值。

7.5.2　包含文件及全局变量

```
#include <windows.h>
#include <math.h>
#include <GL/gl.h>

#if !defined(M_PI)
#define M_PI 3.14159265f
#endif

char *className = "OpenGL";
char *windowName = "Simple Color Index";
int winX = 0, winY = 0;
int winWidth = 300, winHeight = 300;

HDC hDC;
HGLRC hGLRC;
HPALETTE hPalette;

// 用于管理颜色索引的结构
```

```
struct colorIndexState {
    GLfloat amb[3];  /* 环境光颜色/ 斜面底面 */
    GLfloat diff[3];     /* 斜面中面 / 漫反射光颜色 */
    GLfloat spec[3];     /* 斜面底面 / 镜面的射光颜色 */
    GLfloat ratio;      /* 漫反射与镜面反射的比例 */
    GLint indexes[3];     /* 在调色板中的位置 */
};

/* 数组中的每一项都表示在调色板中的每一个颜色值，下标值用来索引每一个颜色集合。
*/
#define NUM_COLORS (sizeof(colors) / sizeof(colors[0]))
struct colorIndexState colors[] = {
    {
        { 0.0f, 0.0f, 0.0f },
        { 0.1f, 0.6f, 0.3f },
        { 1.0f, 1.0f, 1.0f },
        0.75f, { 0, 0, 0 },
    },
    {
        { 0.0f, 0.0f, 0.0f },
        { 0.0f, 0.2f, 0.5f },
        { 1.0f, 1.0f, 1.0f },
        0.75f, { 0, 0, 0 },
    },
    {
        { 0.0f, 0.05f, 0.05f },
        { 0.6f, 0.0f, 0.8f },
        { 1.0f, 1.0f, 1.0f },
        0.75f, { 0, 0, 0 },
    },
};
```

7.5.3 绘制圆环

自定义一个由颜色索引方式表示的圆环，圆环的颜色索引表由上面定义的结构给出。

```
void drawTorus(void)
{
    int numMajor = 32;
    int numMinor = 24;
```

```
        float majorRadius = 0.6f;
        float minorRadius = 0.2f;
        double majorStep = 2.0f*M_PI / numMajor;
        double minorStep = 2.0f*M_PI / numMinor;
        int i, j;

        for (i=0; i<numMajor; ++i) {
         double a0 = i * majorStep;
         double a1 = a0 + majorStep;
         GLfloat x0 = (GLfloat) cos(a0);
         GLfloat y0 = (GLfloat) sin(a0);
         GLfloat x1 = (GLfloat) cos(a1);
         GLfloat y1 = (GLfloat) sin(a1);

         if (i & 1) {
         glMaterialiv(GL_FRONT, GL_COLOR_INDEXES, colors[0].indexes);
         } else {
         glMaterialiv(GL_FRONT, GL_COLOR_INDEXES, colors[1].indexes);
         }

        glBegin(GL_TRIANGLE_STRIP);
        for (j=0; j<=numMinor; ++j) {
           double b = j * minorStep;
           GLfloat c = (GLfloat) cos(b);
           GLfloat r = minorRadius * c + majorRadius;
           GLfloat z = minorRadius * (GLfloat) sin(b);

           glNormal3f(x0*c, y0*c, z/minorRadius);
           glVertex3f(x0*r, y0*r, z);

           glNormal3f(x1*c, y1*c, z/minorRadius);
           glVertex3f(x1*r, y1*r, z);
        }
       glEnd();
      }
}
```

7.5.4 设置投影方式

为了保持物体在场景中的显示比例，需要根据窗口的大小比例，重新调整 glFrustum 函数调用的参数。

```
void setProjection(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (winWidth > winHeight) {
        GLfloat aspect = (GLfloat) winWidth / (GLfloat) winHeight;
        glFrustum(-0.5f*aspect, 0.5f*aspect, -0.5f, 0.5f, 1.0f,
3.0f);
    } else {
        GLfloat aspect = (GLfloat) winHeight / (GLfloat) winWidth;
        glFrustum(-0.5f, 0.5f, -0.5f*aspect, 0.5f*aspect, 1.0f,
3.0f);
    }
    glMatrixMode(GL_MODELVIEW);
}
```

7.5.5 初始化 OpenGL 环境

该函数中完成以下几项工作：清除颜色索引缓存、设置投影方式、设置材质的光亮度、设置光源位置，以及激活光照和深度检测等。

```
void init(void)
{
    GLfloat matShine = 20.00f;
    GLfloat light0Pos[4] = { 0.70f, 0.70f, 1.25f, 0.00f };

    glClearIndex((GLfloat) colors[2].indexes[1]);
    setProjection();
    glTranslatef(0.0f, 0.0f, -2.0f);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, matShine);
    glLightfv(GL_LIGHT0, GL_POSITION, light0Pos);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);
}
```

7.5.6 变换后重绘圆环

该函数将被空闲事件处理程序调用。函数执行时每次绘制一个圆环，然后分别绕 x 轴、y 轴、z 轴分别旋转 x_0，y_0，z_0 度，再将其交换到前台缓存。 x_0、y_0、z_0 用来控制圆环的旋转位置，其值每次分别增加 5，7，9，直到各自大于 360，再分别从 0° 重新开始。这样周而复始，加之双缓存的来回切换，从而形成一个非常平滑的摇摆式的动画效果。

```
void doRedraw(void)
{
    static GLfloat x, y, z;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(x, 1.0f, 0.0f, 0.0f);
    glRotatef(y, 0.0f, 1.0f, 0.0f);
    glRotatef(z, 0.0f, 0.0f, 1.0f);
    drawTorus();
    glPopMatrix();
    SwapBuffers(hDC);
    x += 5.0f;
    if (x > 360.0f) x-= 360.0f;
    y += 7.0f;
    if (y > 360.0f) y-= 360.0f;
    z += 9.0f;
    if (z > 360.0f) z-= 360.0f;
}
```

在空闲事件处理程序中调用上述重绘程序，从而产生圆环反复旋转运动的动画效果。

```
void (*idleFunc)(void);
void redraw(void)
{
    idleFunc = doRedraw;
}
```

7.5.7 窗口改变大小

窗口大小改变时只需重新设置投影矩阵和视口即可。

```
void resize(void)
{
    setProjection();
    glViewport(0, 0, winWidth, winHeight);
```

```
}
```

7.5.8 设置逻辑调色板

设置逻辑调色板就是用颜色表来填充逻辑调色板，从而使之成为系统中可标识的调色板。设置逻辑调色板需要考虑两个问题，即调色板的静态入口和非静态入口的设置。

首先应保证默认的静态入口出现在正确的位置，实现这一点最简单的办法是从当前系统调色板中得到相应的静态入口值，调用 GetSystemPaletteEntries 函数即可。所有非静态入口应初始化为唯一的值，最简单的办法是为每一个非静态入口都设置 PC_NOCOLLAPSE 标志。

```
void setupPalette(HDC hDC)
{
    PIXELFORMATDESCRIPTOR pfd
    LOGPALETTE* pPal
    int pixelFormat = GetPixelFormat(hDC)
    int paletteSize

    DescribePixelFormat(hDC                              pixelFormat
sizeof(PIXELFORMATDESCRIPTOR)  &pfd)
    // 判定是否需要一个调色板，若需要给出其大小
    if (pfd.dwFlags & PFD_NEED_PALETTE ||
        pfd.iPixelType == PFD_TYPE_COLORINDEX) {
        paletteSize = 1 << pfd.cColorBits
        if (paletteSize > 4096) {
            paletteSize = 4096
        }
    } else {
        return
    }

    pPal = (LOGPALETTE*)
malloc(sizeof(LOGPALETTE) + paletteSize * sizeof(PALETTEENTRY))
    pPal.>palVersion = 0x300
    pPal.>palNumEntries = paletteSize

    {
    int numRamps = NUM_COLORS
    int rampSize = (paletteSize . 20) / numRamps
    int extra = (paletteSize . 20) . (numRamps * rampSize)
```

```
    int i  r

    // 从当前系统调色板中拷贝数据来初始化静态入口
GetSystemPaletteEntries(hDC  0  paletteSize  &pPal->palPalEntry
[0])
    // 用所需颜色填充非静态入口
    for (r=0  r<numRamps  ++r) {
        int rampBase = r * rampSize + 10
        PALETTEENTRY *pe = &pPal->palPalEntry[rampBase]
        int diffSize = (int) (rampSize * colors[r].ratio)
        int specSize = rampSize - diffSize

        for (i=0  i<rampSize  ++i) {
        GLfloat *c0  *c1
        GLint a

        if (i < diffSize) {
            c0 = colors[r].amb
            c1 = colors[r].diff
            a = (i * 255) / (diffSize -1)
        } else {
            c0 = colors[r].diff
            c1 = colors[r].spec
            a = ((i -diffSize) * 255) / (specSize -1)
        }

        pe[i].peRed = (BYTE) (a * (c1[0] -c0[0]) + 255 * c0[0])
        pe[i].peGreen = (BYTE) (a * (c1[1] -c0[1]) + 255 * c0[1])
        pe[i].peBlue = (BYTE) (a * (c1[2] -c0[2]) + 255 * c0[2])
        pe[i].peFlags = PC_NOCOLLAPSE
        }

        colors[r].indexes[0] = rampBase
        colors[r].indexes[1] = rampBase + (diffSize-1)
        colors[r].indexes[2] = rampBase + (rampSize-1)
    }

    // 初始化其他非静态入口
    for (i=0  i<extra  ++i) {
```

```
        int index = numRamps*rampSize+10+i
        PALETTEENTRY *pe = &pPal->palPalEntry[index]

        pe->peRed = (BYTE) 0
        pe->peGreen = (BYTE) 0
        pe->peBlue = (BYTE) 0
        pe->peFlags = PC_NOCOLLAPSE
    }
    }

    hPalette = CreatePalette(pPal)
    free(pPal)

    if (hPalette) {
        SelectPalette(hDC  hPalette  FALSE)
        RealizePalette(hDC)
    }
}
```

7.5.9 设置像素格式

```
void setupPixelFormat(HDC hDC)
{
    PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR),     /* 该结构的大小 */
    1,              /* 版本号*/
    PFD_DRAW_TO_WINDOW |        /* 是否支持在窗口中绘图 */
    PFD_SUPPORT_OPENGL |        /* 是否支持 OpenGL */
    PFD_DOUBLEBUFFER,        /* 是否支持双缓存 */
    PFD_TYPE_COLORINDEX,         /* 颜色索引模式 */
    8,              /* 8 位颜色深度 */
    0, 0, 0, 0, 0, 0,        /* 忽略颜色位 */
    0,              /* 没有非透明缓存 */
    0,              /* alpha 位数 (可忽略) */
    0,              /* 累积缓存 */
    0, 0, 0, 0,          /* 累积缓存位数 (可忽略) */
    16,             /* 深度缓存 */
    0,              /* 模板缓存 */
    0,              /* 辅助缓存 */
```

```
        PFD_MAIN_PLANE,             /* 主层 */
        0,                /* 保留 */
        0,  0,  0,              /* 不设置层，可见性及损毁掩模 */
        };
        int SelectedPixelFormat;
        BOOL retVal;

        SelectedPixelFormat = ChoosePixelFormat(hDC, &pfd);
        if (SelectedPixelFormat == 0) {
        MessageBox(WindowFromDC(hDC),
            "ChoosePixelFormat failed\n"
            "This application works best with an 8-bit\n"
            "(256 color) display mode\n",
            "Error",
            MB_ICONERROR | MB_OK);
        exit(1);
        }
        retVal = SetPixelFormat(hDC, SelectedPixelFormat, &pfd);
        if (retVal != TRUE) {
        MessageBox(WindowFromDC(hDC) ,   "SetPixelFormat  failed" ,
    "Error",
            MB_ICONERROR | MB_OK);
        exit(1);
        }
    }
```

7.5.10 主窗口事件处理函数

在接收到 WM_CREATE 事件时，设置 OpenGL 绘制环境，将 OpenGL 绘制描述表与调色板选入的相同设备描述表捆绑到一起。

```
LRESULT APIENTRY WndProc(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    switch (message) {
    case WM_CREATE:
        hDC = GetDC(hWnd);
```

```
        setupPixelFormat(hDC);
        setupPalette(hDC);
        hGLRC = wglCreateContext(hDC);
        wglMakeCurrent(hDC, hGLRC);
        init();
        idleFunc = doRedraw;
        return 0;
    case WM_DESTROY:
        // 结束 OpenGL 绘制，释放相应的动态内存
        idleFunc = NULL;
        if (hGLRC) {
            wglMakeCurrent(NULL, NULL);
            wglDeleteContext(hGLRC);
        }
        ReleaseDC(hWnd, hDC);
        PostQuitMessage(0);
        return 0;
    case WM_SIZE:
        if (hGLRC) {
            winWidth = (int) LOWORD(lParam);
            winHeight = (int) HIWORD(lParam);
            resize();
            return 0;
        }
    case WM_PALETTECHANGED:
        // 如果该窗口不是当前激活窗口，则修改调色板映射
        if (hGLRC && hPalette && (HWND) wParam != hWnd) {
            UnrealizeObject(hPalette);
            SelectPalette(hDC, hPalette, FALSE);
            RealizePalette(hDC);
            redraw();
            return 0;
        }
        break;
    case WM_QUERYNEWPALETTE:
        // 如果该窗口是当前激活窗口，修改调色板映射
        if (hGLRC && hPalette) {
            UnrealizeObject(hPalette);
            SelectPalette(hDC, hPalette, FALSE);
```

```
            RealizePalette(hDC);
            redraw();
            return TRUE;
        }
        break;
    case WM_PAINT:
        // 修改窗口，注意不要使用 BeginPaint 返回的设备描述表，
        // 因为其中不会包含正确的调色板
        if (hGLRC) {
            PAINTSTRUCT ps;
            BeginPaint(hWnd, &ps);
            redraw();
            EndPaint(hWnd, &ps);
            return 0;
        }
        break;
    case WM_CHAR:
        switch ((int)wParam) {
            case VK_ESCAPE:
                DestroyWindow(hWnd);
                return 0;
            default:
                break;
        }
        break;
        default:
            break;
        }
    // 处理其他未处理信息
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

7.5.11 主函数

主函数中首先定义并登记一个窗口类，然后创建一个窗口，将它显示到屏幕上，并强迫它产生重绘，最后定义空闲事件处理程序为 idleFunc。

```
int APIENTRY WinMain(
    HINSTANCE hCurrentInst,
    HINSTANCE hPreviousInst,
```

```
    LPSTR lpszCmdLine,
    int nCmdShow)
{
    WNDCLASS wndClass;
    HWND hWnd;
    MSG msg;
    // 定义并登记一个窗口类
    wndClass.style = CS_HREDRAW | CS_VREDRAW;
    wndClass.lpfnWndProc = WndProc;
    wndClass.cbClsExtra = 0;
    wndClass.cbWndExtra = 0;
    wndClass.hInstance = hCurrentInst;
    wndClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
    wndClass.lpszMenuName = NULL;
    wndClass.lpszClassName = className;
    RegisterClass(&wndClass);
    // 计算窗口的默认大小
    winWidth = GetSystemMetrics(SM_CYSCREEN) / 3;
    winHeight = GetSystemMetrics(SM_CYSCREEN) / 3;

    // 用前面定义的类创建一个窗口
    hWnd = CreateWindow(
    className,          // 窗口类名称
    windowName,         // 窗口标题文字
    WS_OVERLAPPEDWINDOW |     // 窗口风格
    WS_CLIPCHILDREN |
    WS_CLIPSIBLINGS,
    winX, winY,         // 窗口位置
    winWidth, winHeight,     // 窗口大小
    NULL,           // 父窗口句柄
    NULL,           // 菜单句柄
    hCurrentInst,         // 实例句柄
    NULL);           // 没有附加数据
    // 将窗口映射到屏幕上
    ShowWindow(hWnd, nCmdShow);
    // 强迫窗口自身重绘
    UpdateWindow(hWnd);
```

```
        // 处理消息
        while (1) {
            // 当没有任何消息需要处理时，执行空闲任务
            while (idleFunc &&
               PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == FALSE)
            {
                (*idleFunc)();
            }
            if (GetMessage(&msg, NULL, 0, 0) != TRUE) {
                break;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        return msg.wParam;
    }
```

7.5.12 程序执行结果

以下是本程序运行结果，可见，用颜色索引方式也可以得到非常好的绘制效果，其关键在于选择好一套的调色板。

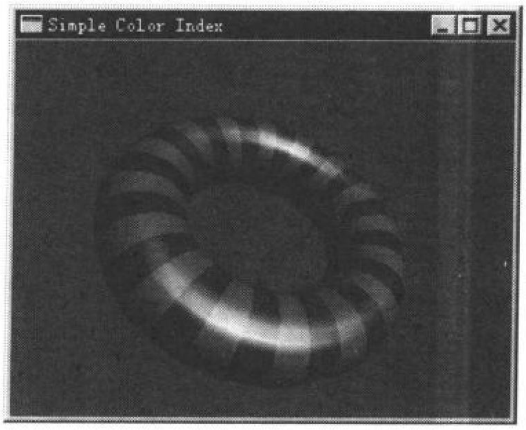

图7.5 简单颜色索引

第 8 章　光照高级编程

本章将讲解以下内容：

- 了解 OpenGL 如何模拟现实的光照效果。
- 定义期望的光源和光照模型。
- 定义被照物体的材质。
- 利用矩阵堆栈控制光源位置。

这些都是进行光照编程的基础知识，介绍完以上内容，将给出一个有趣的发光碰撞小球的例子作为本章内容的总结。

8.1　真实世界与 OpenGL 中的光照

在三维图形应用中显示场景是由其场景中物体自身特性和加在物体上的光照共同决定的。生活中很多时候能感到光照对景物的影响，例如，在阳光普照的晴天，人们看到的大海宛如一块明丽的蓝宝石，而在乌云密布的阴天人们看到的大海则只是灰蒙蒙一片。如果没有光照，大部分物体看起来根本没有三维效果。入射光和物体对光的吸收和反射决定了所看到的景色，这也就是 OpenGL 对光照计算的部分依据。如图 6.1 所示是一个球在光照和无光照情况下的样子。

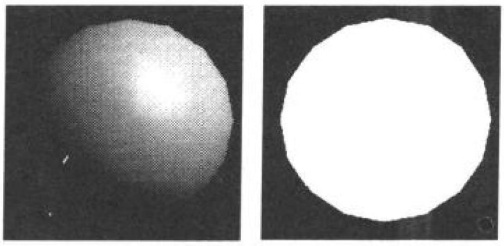

图8.1　有光照和无光照球面的比较

从图中可以看出，没有光照的球面看起来和一个二维的圆盘没有区别，只有加了光照的物体才能显现出三维效果，这充分表明光照和物体的相互作用是绘制三维场景的关键。用户可以通过在 OpenGL 中调整光照和物体在场景中的关系产生出各种各样的效果。这是本章着重讨论的内容，包括 OpenGL 的光照模型及众多的参数。本章还将从数学上介绍光照对颜色的影响。

如前所述，最终的光照效果是光线与表面共同作用的结果。在 OpenGL 中，光线被假定成红、绿、蓝 3 种原色光的组合，因此，光源的颜色就由它所发出的这 3 种原色光的含量决定，而物理表面的颜色则由其反射的这 3 种光的含量决定。OpenGL 的光照模型尽管

只是一个近似，但很管用，且计算速度很快。如果用户需要一个更精确的模型，只好自己编程计算，不过模型越精确，计算速度越慢。

在场景中，OpenGL 可以放置若干个光源，每个光源都可以单独打开或关闭。照在物体上的光线有些是直接来自光源，而另一些是光源的光经过若干次反射而来。这些多次反射的光被称为环境光(Ambient Light)，它们已经变得很“散”，以致于无法确定其初始方向，不过当某个光源(产生它们的光源)被关闭后，它们也就消失了。

不同的物理表面对光线有不同的反射特性，那些通常看起来很有光泽的表面能够将入射光很好地向某一特定方向反射，而另外一些表面则将入射光均衡地散射向各个方向。大部分表面的特性是介于这两者之间的。当然，有些表面也会自己发光，比如汽车前灯。

8.1.1 光照分量

OpenGL 对光照效果的计算是由 4 个独立的部分叠加而成的，即射光(emitted)、环境光(ambient)、漫反射光(diffuse)、镜面光(specular)。

发射光是最简单的，它从物体发出且不受任何其他光源的影响。

环境光是从光源发出经过多次反射形成的，它均匀地从周围入射至物体表面并向各个方向等量反射。

漫反射光是直接从特定光源入射到物体表面并向各个方向等量反射的光，入射角越小看起来越亮，同时从各个角度看亮度是一样的。

镜面光也是直接从特定光源入射的，但在反射时只向特定的方向反射(遵从镜面反射定理)。一束平行激光在高质量的镜面上几乎可以 100%反射。表面光泽的金属或塑料都有很高的镜面反射成份，而粉笔或地毯光泽度差的物品则几乎没有。

尽管同一光源射出的光频率分布是一定的，但照在不同的表面上产生的环境光、漫反射光和镜面光分量却可能不同。比如，白光照在红色墙壁上，散射的光就会接近红色。OpenGL 允许对各种反射光分量的 RGB 值分别以调整。

8.1.2 材质颜色

OpenGL 中近似认为，材质的颜色由反射光中的红、绿、蓝 3 色所占的百分比决定。比如，一个理想的红色球将反射所有红光并吸收所有绿和蓝光。这样的球在白光(含红、绿、蓝 3 色光)和纯红光照射下的表现是一致的——都是红色球，然而，在不含红光成份的光(如纯绿光)的照射下则呈现为黑色(因为没有光被反射)。

与光线对应，材质具有独立的环境反射、漫反射和镜面反射颜色成份，分别决定了材质对环境光、漫反射光和镜面光的反射能力。环境反射和漫反射决定了物体的颜色，这两种成份很相象，甚至可以当作一种来看。镜面反射色通常为白色或灰色，因此镜面反射的高光(highlights)通常是光源的镜面光成份的颜色和强度决定。比如一束白光照射在一个有光泽的红色金属球上，球体的大部分将表现为红色，而高光处则为白色。

8.1.3 光线与材质的 RGB 值

光线颜色的 RGB 值定义与材质的 RGB 值定义有所不同。

对光线来说，RGB 值的大小对应于该色光强度与其最高强度的百分比。比如，一束光的 RGB 值都是 1.0，它就是最强的白光；如果这些值都为 0.5，则整束光的颜色还是白色，只不过强度减半，看上去是灰色。如果 R=G=1.0 而 B=0.0，就是最强的黄光。

对材质来说，RGB 值的大小对应于材质对该色光的反射百分比。比如，某材质的 R=1.0，G=0.5，B=0.0，则它反射全部红光，一半绿光，不反射蓝光。换句话来说，如果一束光的 RGB 值为(LR， LG， LB)，被照材质的 RGB 值为(MR， MG， MB)，则在不考虑其他反射效果的情况下，眼睛所看到的颜色由(LR*MR， LG*MG， LB*MB)决定。

与此类似，如果到达眼睛的两束光的 RGB 值分别为(R1， G1， B1)和(R2， B2， G2)，则 OpenGL 将对应的值相加，得到(R1+R2，G1+G2，B1+B2)。不管哪一个相加结果大于 1(表明，这样的光强设备不能显示)，都视为 1。

8.2 简单示例

向场景中加入光照，需执行以下步骤：

(1) 定义所有物体每个顶点的法向量方向。这些法向量决定了物体表面与光源的相对取向。

(2) 建立、选择并放置一个或多个光源。

(3) 建立并选择一个光照模型，它决定了全局环境光照的程度以及视点的有效位置(用于光照计算)。

(4) 定义场景中物体的材质。

例 8.1 显示一个独立光源照射下的球体，如图 8.1 所示。

例 8.1 光照下的球体

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
void myinit(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
```

```
        glEnable(GL_LIGHT0);
        glDepthFunc(GL_LEQUAL);
        glEnable(GL_DEPTH_TEST);
    }
    void display(void)
    {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        auxSolidSphere(1.0);
        glFlush();
    }
    void myReshape(GLsizei w, GLsizei h)
    {
        glViewport(0, 0, w, h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if (w <= h)
            glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
                1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
        else
            glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
                1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }

    int main(int argc, char** argv)
    {
        auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
        auxInitPosition (0, 0, 500, 500);
        auxInitWindow (argv[0]);
        myinit();
        auxReshapeFunc (myReshape);
        auxMainLoop(display);
    }
```

myimit()函数中调用了与光照有关的函数，下面简单讨论它们的用法，在后面的章节中还会仔细研究。需要注意的一点是在例 8.1 中，使用的是 RGBA 颜色模式，而不是颜色索引模式(color.index)。OpenGL 对这两种模型的光照计算是不同的，而且颜色索引模式的光照表现力有限，因此，比较而言 RGBA 更受推崇的，本章中的例子都使用该模式。如需对颜色索引模式作进一步的了解请参阅 8.7 “颜色索引模式中的光照” 一节。

场景中物体的材质特性决定了光线是如何反射的从而表现出场景中的物体是用什么材料做的。而入射光线与物体材质表面的作用是复杂的，所以通过设置物体的材质特性而使场景中的物体“看起来像那么回事”需要很高的技巧。需要制定一种材质的环境光反射色、漫射光反射色、镜面光反射色(这里说 XX 反射色的意思是指对某种光的某个颜色分量的反射能力)以及它的光泽度。在这个例子中，最后两个特性——镜面光反射颜色和光泽度作了明确指定(通过使用 glMaterialfv()函数)。在 8.5“定义材质特性”一节中会给出所有的材质特性参数并给出例子。

注意：在写光照程序时，可以修改一些参数值，参数使用默认值。当然，不要忘记使定义的光源生效(即调用 glEnable()函数)，同时允许光照计算。最后，记住可以使用显示列表以提高在改变光源位置时的计算。

8.3 创建光源

光源有如下属性：颜色、位置和方向。下面讨论如何控制这些属性。用来设置光源属性的函数是 glLight*()，它有 3 个参数，分别代表光源标识、需设置的属性和期望的属性值。函数形式为：

```
void glLight{if}[v](GLenum light, GLenum pname, TYPE param);
```

light 是光源的标识，例如：GL_LIGHT0，GL_LIGHT1，…GL_LIGHT7。一般地，GL_LIGHTi 代表第 i 号光源，pname 代表需设置的属性，其枚举值和对应的属性意义见表 8.1。param 代表需要给 pname 代表的属性设置的值。

注意：TYPE 是 param 的类型，它由 glLight 函数名后面跟的字母决定，i 就是 int，f 就是 float，v 就是 vectro，可选，选了说明 param 代表指向一组参数的指针，不选则说明是单一参数。

表8.1 glLight*()的pname参数及其默认值

参数名称	默认值	说 明
GL_AMBIENT	0，0，0，1	RGBA 模式的环境光
GL_DIFFUSE	1，1，1，1	RGBA 模式的漫反射光
GL_SPECULAR	1，1，1，1	RGBA 模式的镜面光
GL_POSTION	1，0，1，0	光源位置齐次坐标(x， y， z， w)
GL_SPOT_DIRECTION	0，0，-1	聚光灯聚光方向矢量
GL_SPOT_EXPONENT	0	聚光灯聚光指数
GL_SPOT_CUTOFF	180	聚光灯聚光发散半角
GL_CONSTANT_ATTENUATION	1	常数衰减因子
GL_LINER_ATTENUATION	0	线性衰减因子
GL_QUADRATIC_ATTENUATION	0	平方衰减因子

表 8.1 中 GL_DIFFUSE 和 GL_SPECULAR 的默认值只对 GL_LIGHT0 有效，对于其他的光源，它们的默认值都是（0.0， 0.0， 0.0， 1.0）。

下面是使用 glLight*()的一个例子：

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

数组被用来给参数赋值，通过反复调用 glLightfv()给不同的参数赋值。在这个例子里，前 3 条语句是多余的，因为它们设置的分别是那 3 个参数的默认值。

记住要用 glEnable 打开每个光源。要了解更多的相关操作，请参阅 8.4.4“启动光照”一节。下面的部分将解释 glLight*()的所有参数和它们可能的取值。这些参数和定义与光照模型和物体材质特性的参数息息相关。请参阅 8.4“选择光照模型”和 8.5“定义材质特性”一节以获取更多的知识。在 8.6“光照中的数学”一节中，将从数学的角度解释这些参数是如何相互作用的。

8.3.1 颜色

OpenGL 允许对光源的 3 个与颜色相关的参数进行调整，它们是：GL_AMBIENT，GL_DIFFUSE 和 GL_SPECULAR。

GL_AMBIENT 参数代表场景中某光源的环境光的 RGBA 各个分量强度。如表 8.1 所示，默认状态下是没有环境光的，因为 GL_AMBIENT 是(0.0，0.0，0.0，1.0)。例 8.1 使用的正是该值。如果程序将环境光设为蓝光，应该设置如下：

```
GLfloat light_ambient[] = { 0.0, 0.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

GL_DIFFUSE 可能是与“光的颜色”联系最大的参数了，它定义了特定光源提供的漫反射光的 RGBA 颜色值。默认状态下，光源 GL_LIGHT0 的 GL_DIFFUSE 值是（1.0，1.0，1.0，1.0)，即产生明亮的白光，而其他光源(GL_LIGHT1 至 GL_LIGHT7)的默认值都是（0.0，0.0，0.0，0.0)。

GL_SPECULAR 参数影响物体上的高光点颜色。实际中的典型例子如玻璃瓶子上的高光点显示的就是照在上面的光的颜色。因此，如果想获得逼真的效果，应该将 GL_SPECULAR 的值设得和 GL_DIFFUSE 一样。所有光源的 GL_SPECULAR 参数默认值和 GL_DIFFUSE 默认值也是一样的。

8.3.2 位置与衰减

光源的位置有 2 种，一种离场景无限远，另一种在附近。远的称为方向光源，它射到物体上的光可以认为是平行的，现实生活中，太阳就是这样的光源。近的称为位置光源，它的具有位置决定了它对场景的作用效果，尤其是决定了光线的投射方向，台灯就是位置光源。

例 8.1 使用的是方向光源：

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

如上所示，GL_POSITION 参数是一个 4 个值的向量(x, y, z, w)。如果最后一个值 w 是 0，则相应的光源为方向光源，由(x，y，z)决定其方向。这个方向将被模型视图矩阵转换到视点坐标系中，就好象它是一个被标准化的向量一样。GL_POSITION 的默认值为(0, 0, 1, 0)，指向 Z 轴的负方向。

如果 w 是非零值，光源就是位置光源。而(x，y，z)代表光源位置的齐次坐标。该位置将被透视模型矩阵转换到视点坐标系中。在默认状态下，光源的光是向各个方向发出的，不过也可以定义一个聚光灯使其发射锥光，这将在 8.33“聚光灯”一节中解释。

注意： *具有光滑阴影效果的多边形表面的各部分颜色是由各顶点的颜色计算结果决定的。因此，要避免在局部光源附近使用大多边形——如果光源位置在多边形的中部，各顶点会因为距离光源太远而接收不到足够的光照，从而整个多边形看起来要比你相象的要暗。要解决这个问题，就得把大多边形分解成许多小块。*

在现实世界中，光线的强度会随着传播距离的增加而减弱。方向光源是从无限远处射来的，对它谈距离衰减意义不大，所以衰减计算对方向光源是无效的。然而，对位置光源则必须考虑这个问题。OpenGL 通过给光源的光乘上一个因子来达到衰减的目的。

$$\text{衰减因子} = \frac{1}{k_c + k_l d + k_q d^2}$$

其中，d 为光源到顶点的距离，k_c 为 GL_CONSTANT_ATTENUATION，k_l 为 GL_LINEAR_ATTENUATION，k_q 为 GL_QUADRATIC_ATTENUATION。默认状态下，k_c 为 1.0，k_l 和 k_q 都为 0，当然也可以设置其他值，比如：

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

注意： *环境光、漫反射光和镜面光都会被衰减，只有发射光和全局环境光不被衰减。*

8.3.3 聚光灯

上节提到，可以将一个位置光源定义成一个聚光灯，也就是说可以将光的发射形状调整为圆锥形。创建聚光灯，需要定义锥光的跨度(记住，因为聚光灯是位置光源，还需要为

它定义一个位置。当然也可以将方向光源定义为聚光灯，语法上没错，但无意义)。定义锥光的散射角(即圆锥中轴和边的夹角)，需要使用 GL_SPOT_CUTOFF 参数，圆锥的顶角是该值的 2 倍，见图 8.2。

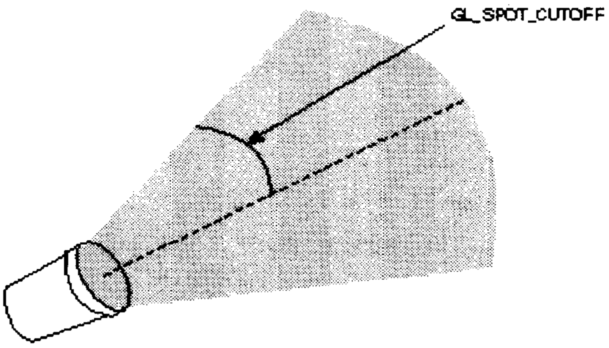

图8.2 GL_SPOT_CUTOFF参数示意

光线不能超越圆锥的边界。默认时没有聚光灯效果，因为 GL_SPOT_CUTOFF 参数被默认为 180°，也就意味着光线是向各个方向发射的(此时圆锥的顶角为 360°，已经不是"锥"了)。GL_SPOT_CUTOFF 一般的取值范围是[0.0，90.0](或者就是取 180)。下面代码将散射角置为 45°。

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

还需给出聚光灯的方向，即圆锥轴的方向。

```
GLfloat spot_direction[] = {-1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

方向是用齐次坐标表示的，默认状态下为(0，0，1)，所以如果不显示设置 GL_SPOT_DIRECTION 的值，聚光灯光线将射向 z 轴负向。同时，这个方向将被透视模型矩阵按其标准化后的值转换到视点坐标系中。(参见 8.3.5"控制光源的位置和方向"一节以了解更多的有关位置转换的知识)

除了聚光灯的散射角和方向外，还可以控制光线在圆锥内的强度分布。有 2 个办法，首先，可以先设置光线衰减因子，前面提到过，它将与光强相乘；也可以设置聚光指数，即 GL_SPOT_EXPONENT 参数，它决定了光线的集中性，默认值为 0。光线在圆锥的中心最强，越靠边界越弱，其衰减与该光线和圆锥轴夹角的 COS 值的 GL_SPOT_EXPONENT 参数次方成正比。因此聚光指数越大，锥光的集中性越好。请参阅 8.6"光照中的数学"一节以了解光强计算的细节。

8.3.4 多光源

如前所述，可以在场景中至少使用 8 个光源(当然可以更多，这取决于用户的需要)。因为 OpenGL 需要对每个顶点从所有光源接收的光进行计算，所以增加光源的数目无疑会使系统处理速度下降。用来代表每个光源的常量为 GL_LIGHT0，GL_LIGHT1，GL_LIGHT2，GL_LIGHT3 等。前面已经讨论如何对与 GL_LIGHT0 相关的参数进行设置。

如果要增加光源，则必须对其参数进行设置。不过，其他光源的默认参数与 GL_LIGHT0 的是不同的，如表 8.2 所示。下面的代码定义了一个衰减的白色聚光灯。

```
GLfloat light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_position[] = { -2.0, 2.0, 1.0, 1.0 };
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);
glEnable(GL_LIGHT1);
```

如果将这些代码加入例 8.1，那么其中的球体将被两个光源照射，一个是方向型的，另一个是位置型的。

可以尝试修改例 8.1，将原来的白色方向光源修改为一个位置型有色光源，再加入一个有色聚光灯。

提示：可以参考上面的代码感觉一下这 2 种改动对整体效果的影响。

8.3.5 控制光源位置和方向

OpenGL 对光源位置和方向的处理与它对基本几何形体的处理是类似的。换句话说，光源与基本的几何形体有着相同的矩阵变换。当使用 glLight*()来指定光源的位置和方向时，这些位置和方向就通过当前的透视模型矩阵转换到视点坐标系中。这说明可以通过修改当前透视模型矩阵栈中的内容来控制光源的位置和方向(投影矩阵不会对光源的位置和方向产生影响)。本节将介绍如何通过修改程序中对光源位置的设置来获得 3 种不同的效果，这是与模型和透视变换相关的。这 3 种光源是：

- 固定位置光源。
- 环绕固定物体运动的光源。
- 和视点一起运动的光源。

例 8.1 作为最简单的例子，演示了固定位置光源。要达到这个效果，你只需要在使用了透视或(和)模型变换后设置光源位置。下面是 myinit()和 myReshape()函数中的相关语句：

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
```

```
if (w <= h)
glOrtho (-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
glOrtho (-1.5*w/h, 1.5*w/h, -1.5, 1.5, .10.0, 10.0);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity();
/* later in myInit() */
GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

视窗和投影矩阵被首先确定。然后，单位矩阵被调入作为透视模型矩阵，接下来设置光源位置。因为使用了单位矩阵，所以光源的初始位置(1.0，1.0，1.0)和它相乘并没有变化。在这之后，光源位置和透视模型矩阵都没有改变，所以光源还放在(1.0，1.0，1.0)的位置。

现在假设用户要旋转或移动光源的位置使其相对一个静态的物体运动。一个办法是在模型变换后设置光源位置，光源位置改变的效果是通过模型变换矩阵的改变来实现的。此外，还可以在程序开头使用上面的 init()函数，然后，在程序消息循环里加入想要的模型变换(利用透视模型矩阵)从而确定光源的新位置。下面是一段可能的代码：

```
void display(GLint spin)
{
GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glTranslatef(0.0, 0.0, -5.0);
glPushMatrix();
glRotated((GLdouble) spin, 1.0, 0.0, 0.0);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glPopMatrix();
auxSolidTorus(0.275, 0.85);
glPopMatrix();
glFlush();
}
```

display()函数重画场景，重画时光源好像是绕静态圆环面转过了 spin 代表的角度。请注意 glPushMatrix()和 glPopMatrix()函数，它们是用来隔离透视和模型变换的，之所以要隔离是因为需要画的圆环面是不旋转的，即画它时只受到 glTranslatef(0.0，0.0，-5.0)；这条语句的影响。在本例中视点保持不变，所以当前的变换阵被压入堆栈，然后由 glTranslatef()完成透视变换(位移)，将现在的变换矩阵压入堆栈，再由 glRotated()完成模型变换(旋转)。此时设置新的光源位置，由于此位置是在旋转后的坐标系中定义的，所以变换回原坐标系中后就好像光源被旋转了一样(原坐标也就是视点坐标系，与之对应的变换阵一直被压在栈里)。在光源位置定义完毕后，与透视变换对应的矩阵被弹出，此时再画圆环面，就像前面

所说的那样，转换到原坐标系中，圆环面只有位置变换而无旋转变换，所以看起来就有了光源环绕圆环面旋转的效果。

为了创建和视点一起移动的光源，需要在透视模型变换之前设置光源位置，这样，视点和光源就会按相同的方式一起变换。可以在 myinit()函数中，作一些小改动来达到这个效果：

```
GLfloat light_position[] = { 0.0, 0.0, 1.0, 1.0 }; // 这样光源好
像是放在视点的位置上
glViewport(0, 0, w.1, h.1);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
glMatrixMode(GL_MODELVIEW);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

然后对 display()函数修改如下：

```
void display(GLint spin)
{
    glClear(GL_COLOR_BUFFER_MASK | GL_DEPTH_BUFFER_MASK);
    glPushMatrix();
    glTranslatef (0.0, 0.0, -5.0);
    glRotatef ((GLfloat) spin, 1.0, 0.0, 0.0);
    auxSolidTorus(0.275, 0.85);
    glPopMatrix();
    glFlush();
}
```

当被照的圆环面重画时，光源和视点都被移动了 spin 角度。因为是在视点的坐标系中，所以看到的效果是一个旋转的圆环面被固定的视点的光源照射的效果。

可以尝试一下修改，下面的例子 8.1：

- 让光源只是相对物体移动而不是环绕其转动。

提示：在 display()中用 glTranslated()代替 glRotated()，同时找一个合适的参数代替 spin。

- 改变光源衰减模式，使光源远离物体时物体上的光照会减弱。提示：加入 glLight*()设置相应参数。

例 8.2 通过模型变换移动光源 movelight.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
static int spin = 0;
void movelight (AUX_EVENTREC *event)
{
    spin = (spin + 30) % 360;
```

```
}
void myinit (void)
{
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}
void display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glTranslatef (0.0, 0.0, .5.0);
    glPushMatrix ();
    glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
    glRotated (0.0, 1.0, 0.0, 0.0);
    glLightfv (GL_LIGHT0, GL_POSITION, position);
    glTranslated (0.0, 0.0, 1.5);
    glDisable (GL_LIGHTING);
    glColor3f (0.0, 1.0, 1.0);
    auxWireCube (0.1);
    glEnable (GL_LIGHTING);
    glPopMatrix ();
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();
    glFlush ();
}
void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
```

```
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, movelight);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

8.4 选择光照模型

OpenGL 的光照模型概念有如下 3 个方面：

- 全局环境光强度。
- 视点位置在场景内还是无限远处。
- 对物体表面的正反面光照计算是否不同。

本节介绍如何设置光源，以及如何启用光照，即告诉 OpenGL 需要对相应光照进行计算。

8.4.1 全局环境光

如前所述，每个光源都对场景的环境光有贡献。同时，还有一种环境光不是由光源提供的，这就是全局环境光。 设置全局环境光的 RGBA 强度，需要按下面的方式使用 GL_LIGHT_MODEL_AMBIENT 参数：

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

本例中 lmodel_ambient 的参数值都是 GL_LIGHT_MODEL_AMBIENT 的默认值。因为这些值会产生少量的环境光，所以即使不定义其他光源仍可看到场景中的物体。

8.4.2 局部和无穷远视点

视点位置影响镜面反射导致的高光点计算。具体地讲，高光点的强度由 3 个因素决定：被照顶点的法向量、从光源到顶点的光线方向、视点和顶点的连线方向。有一点要明白，使用各种光照命令后，实际上移动的并不是视点，只是改变了投影方式，使视点看起来移动了。不过，在光照计算中，很多地方是假设光源位置改变的。

对于无穷远视点来说，它和某顶点的连线方向是个常数。虽然，局部视点的效果更趋于真实，但由于它要对每个顶点的方向分别计算，就大大增加了程序的负担，所以默认状态下，视点都是在无限远处的。通过下面的语句可以把它变成局部的：

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

这条语句把视点放到了视点坐标系的(0, 0, 0)位置。如果想变回无限远视点，只要把最后一个参数改成 GL_FALSE 就行了。

8.4.3 双面光照

光照计算是对所有多边形表面进行的，不管是正面还是反面。由于通常设置的都是正面的光照性质，背面的光照效果往往与正面不同。在例 8.1 中，被照物体是球面，看到的永远是其正面，也就是球的外表面，背面的光照效果根本看不到，也就无所谓了。如果切开球面，就会看到内侧。这时如果希望里面的光照效果和外面定义的一致，同时给内表面贴上一种不同的材质，启起用双侧光照，语句如下：

```
glLightModeli(LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

通过这个语句 OpenGL 把正面顶点的法向量反过来作为背面的法向量，这样，所有多边形的光照效果就一样了。

关闭双侧光照，只需把前面语句中的 GL_TRUE 换成 GL_FALSE 即可。除此之外，还可以通过 glFrontFace()命令设置哪个表面是正面。

8.4.4 启用光照

在 OpenGL 中需要显式地启用(或取消)光照计算。如果没有和用光照，当前的颜色只是简单地被映射到当前顶点上，不会进行任何关于法向量、光源、光照模型和材质的计算。启用光照命令如下：

```
glEnable(GL_LIGHTING);
```

取消光照命令为 glDisable(GL_LIGHTING);。

设置了定义的光源后，必须显式地打开它们。在例 8.1 中只使用了一个光源 GL_LIGHT0。打开它的语句为：

```
glEnable(GL_LIGHT0);
```

8.5 定义材质特性

本节介绍如何定义物体的材质：环境反射色、漫反射色和镜面反射色，光泽度，以及出射光色。在光照和材质计算中用到的公式会在 8.6“光照中的数学”一节中介绍。大部分的材质特性参数与在建立光源时用过的参数相似，设置它们的机制也差不多，只不过这回使用的命令是 glMaterial*()罢了。其完整形式为：

```
void glMaterial{if}[v](Glenum face, GLenum pname, TYPE param);
```

face 参数可以是 GL_FRONT， GL_BACK， 或 GL_FRONT_AND_BACK，分别代表材质应该贴到物体的哪个表面。材质特性的名称通过 pname 设置，特性值由 param 参数确定。param 可以是单值或是指向一组参数的指针(如果函数把向量作为表示形式，函数名称后面一定有“v”字符)。非向量表示形式(函数名后面没有“v”)只用来设置 GL_SHININESS

参数。pname 可以表示的参数意义和对应默认值见表 8.2。

表8.2 glMaterial*()的pname参数及其默认值

pname	默认值	说明
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	材料环境光反射色
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	材料漫反射光反射色
GL_AMBIENT_AND_DIFFUSE		设置上面 2 项为相同值
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	材料镜面光反射色
GL_SHININESS	0.0	材料反射指数
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	材料发射光色
GL_CO8.4LOR_INDEXES	(0, 1, 1)	环境、漫射、镜面光反射颜色索引

在“选择光照模型”一节中讨论过，可以为物体的正反表面设置不同的光照效果。如果反面的确可以被看到，可以通过 glMaterial*()的 face 参数来使正反表面具有不同的材质特性。

注意：大部分材质特性的设置值是(R，G，B，A)颜色。不管在其他参数中 alpha 的含义，相对某个特定的顶点，它表示该顶点的漫反射光反射色的 alpha，也就是给 GL_DIFFUSE 参数设置值的 alpha 值。同时，任何 RGBA 模式的材质特性使用的都不是颜色索引模式。

8.5.1 漫射和环境反射

通过 glMaterial*()设置 GL_DIFFUSE 和 GL_AMBIENT 参数可以影响物体反射的漫射光和环境光的颜色。漫射光是影响物体颜色的最重要的因素，其强度取决于入射光的颜色以及入射光和顶点法向量的夹角(当入射光垂直于表面时漫射光最强)。视点位置对漫射光无影响。

环境光影响物体的总体色彩。因为当物体被光源直接照射时眼睛看到的主要是漫射光，所以只有当没有直接照明时环境光在物体上的效果才会被明显的看到。物体的环境反射光由全局环境光和各光源的环境光成分决定。与漫射光类似，环境光也不受视点位置的影响。对于现实的物体来说，环境光和漫射光的颜色基本是一致的。因此，OpenGL 提供了一个方法，通过它可以方便地使用 glMaterial*()一次将漫射光和环境光参数设置为相同的值：

```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK , GL_AMBIENT_AND_DIFFUSE , mat_amb_
diff);
```

在本例中，RGBA 色为(0.1，0.5，0.8，1.0)——一种深蓝色，这条语句将物体正反表面的环境光和漫射光色都设置为这种颜色。

8.5.2 镜面反射

物体上的镜面反射导致高光点。与环境光和漫射光不同，镜面反射的光强与视点的位置关系密切——从反射角的方向看最亮。想理解这一点，可以想象在阳光下看一个金属球的情形。移动脑袋的时候，高光点也会跟着移动一些。而移动超过一定限度时，你就完全看不到高光点了。

OpenGL 允许你设置镜面反射高光点的 RGBA 颜色(通过 GL_SPECULAR 参数)，并控制高光点的大小和明亮程度(通过 GL_SHININESS 参数)。GL_SHININESS 的取值范围是[0.0，128.0]。值越高，高光点越小，越亮(聚焦越好)。想了解高光点的计算请参阅 8.6 “光照中的数学” 一节。

8.5.3 发射光

通过给 GL_EMISSION 参数设置一个 RGBA 值，可以使物体看起来好像在发射那种颜色的光。因为大部分现实世界的物体(除了光源)都不发射光，所以最可能使用这个功能来模拟一盏灯以及场景中的其他光源。例如，想设置球体具有绿色的 GL_EMISSION，可以使用下面的语句：

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

这些球体微微有些荧光灯泡的感觉，不过它们在场景中不会起到光源的作用。要想真正有灯泡的效果，必须在球体的位置再定义一个位置光源。

8.5.4 改变材质特性

例 8.1 给场景中物体(球体)的所有顶点都赋与了相同的材质。有时，需要对同一个物体的不同顶点赋与不同的材质。尤其是场景中有不止一个物体，而每个物体都有不同的材质特性。例 8.3 给出了这样一个例子(这些语句写在 display()函数中)：

例 8.3 使用不同的材质 material.c

```
GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 };
GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat no_shininess[] = { 0.0 };
GLfloat low_shininess[] = { 5.0 };
GLfloat high_shininess[] = { 100.0 };
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
// 第一行第一列的球体，只有漫反射，没有环境光或镜面反射
glPushMatrix();
glTranslatef (-3.75, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
auxSolidSphere();
glPopMatrix();
// 第一行第二列的球体，有漫反射和镜面反射，低光泽度，无环境光
glPushMatrix();
glTranslatef (-1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
auxSolidSphere();
glPopMatrix();
// 第一行第三列的球体，有漫反射和镜面反射，高光泽度，无环境光
glPushMatrix();
glTranslatef (1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
auxSolidSphere();
glPopMatrix();
// 第一行第四列的球体，有漫反射和出射光，无环境光和镜面反射
glPushMatrix();
glTranslatef (3.75, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
auxSolidSphere();
```

```
glPopMatrix();
```

如上所示，glMaterial*()被反复调用以设置各球期望的材质特性。只有一个特性需要修改时才应该调用一个 glMaterialf*()。举个例子，第二和第三球的环境光和漫射特性与第一个球的一致，所以这些特性不用重新设置。因为 glMaterialf*()的调用次数影响到系统消耗，所以对材质特性的修改越少越好。

另一种减少修改材质特性的消耗的办法是使用 glColorMaterial()，其形式为

```
void glColorMaterial(Glenum face, GLenum mode);
```

该命令使材质的特性色和当前颜色(由 glColor*()设置)一致。face 参数可以是 GL_FRONT，GL_BACK，或 GL_FRONT_AND_BACK(默认值)，mode 参数可以是 GL_AMBIENT，GL_DIFFUSE，GL_AMBIENT_AND_DIFFUSE(默认值)，GL_SPECULAR，或 GL_EMISSION。face 代表哪个或哪些表面的材质特性色需要改变为当前颜色，mode 代表哪些材质特性色需要改变。OpenGL 不允许给同一面设置不同的 mode。

使用了 glColorMaterial()命令后，还需要调用 glEnable(GL_COLOR_MATERIAL)命令使之有效。下面，就可以使用 glColor*()来改变当前颜色并影响后面所画的物体的颜色了(使用 glMaterial*()可以改变其他材质特性)。下面是一段例子：

```
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
glColor3f(0.2, 0.5, 0.8);
/* 画一些东西 */
glColor3f(0.9, 0.0, 0.2);
/* 再画一些东西 */
glDisable(GL_COLOR_MATERIAL);
```

需要改变场景中的大部分顶点的某个单独材质特性时，就应该使用 glColorMaterial()，这样做速度最快。如果改变的是不止一个特性，则需要使用 glMaterial*()。当不再需要 glColorMaterial()功能时，要用 glDisable(GL_COLOR_MATERIAL)使之无效，以减少系统消耗。

例 8.4 给出了一个利用 glColorMaterial()改变材质参数的互动式程序。按动 3 个鼠标键可以将物体的散射光色改变为不同颜色。

例 8.4 使用 glColorMaterial()：colormat.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
GLfloat diffuseMaterial[4] = { 0.5, 0.5, 0.5, 1.0 };
void myinit(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

```
    glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
}

void changeRedDiffuse (AUX_EVENTREC *event)
{
    diffuseMaterial[0] += 0.1;
    if (diffuseMaterial[0] > 1.0)
    diffuseMaterial[0] = 0.0;
    glColor4fv(diffuseMaterial);
}
void changeGreenDiffuse (AUX_EVENTREC *event)
{
    diffuseMaterial[1] += 0.1;
    if (diffuseMaterial[1] > 1.0)
    diffuseMaterial[1] = 0.0;
    glColor4fv(diffuseMaterial);
}
void changeBlueDiffuse (AUX_EVENTREC *event)
{
    diffuseMaterial[2] += 0.1;
    if (diffuseMaterial[2] > 1.0)
    diffuseMaterial[2] = 0.0;
    glColor4fv(diffuseMaterial);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    auxSolidSphere(1.0);
    glFlush();
}
void myReshape(GLsizei w, GLsizei h)
{
```

```
        glViewport(0,  0,  w,  h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if (w <= h)
            glOrtho (-1.5,  1.5, -1.5*(GLfloat)h/(GLfloat)w,
                    1.5*(GLfloat)h/(GLfloat)w, -10.0,  10.0);
        else
            glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
                1.5*(GLfloat)w/(GLfloat)h, -1.5,  1.5, -10.0,  10.0);
            glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }

    int main(int argc,  char** argv)
    {
        auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
        auxInitPosition (0,  0,  500,  500);
        auxInitWindow (argv[0]);
        myinit();
        auxMouseFunc(AUX_LEFTBUTTON,  AUX_MOUSEDOWN,
                    changeRedDiffuse);
        auxMouseFunc(AUX_MIDDLEBUTTON,  AUX_MOUSEDOWN,
                    changeGreenDiffuse);
        auxMouseFunc(AUX_RIGHTBUTTON,  AUX_MOUSEDOWN,
                    changeBlueDiffuse);
        auxReshapeFunc(myReshape);
        auxMainLoop(display);
    }
```

可以尝试一下修改例 8.3，修改全局环境光。

提示： 改变 GL_LIGHT_MODEL_AMBIENT 参数值。改变散射光、环境光和镜面反射光，光泽度指数，以及发射光。使用 glMaterial*()命令，但注意不要调用次数过多。使用双侧光照，并加入一个任意的裁剪面，这样你可以看到一行或一列的球体的内部光照情况。

注意： 可以打开 GL_LIGHT_MODEL_TWO_SIDE 开关，设置期望的材质特性，并加入裁剪面。去掉所有的 glMaterial*()语句，多使用一些高效率的 glColorMaterial()语句达到同样的光照效果。

8.6　光照中的数学

本节将给出一些公式，它们是 OpenGL 用来计算光照下的 RGBA 模式颜色的。(在 8.7 “索引颜色模式中的光照”一节中，可以找到关于索引颜色的相应计算)。如果想通过尝试的办法获得理想的光照效果，那么没有必要阅读本节。而且即使在读完本节后，还是需要通过尝试才能真正掌握光照的应用。当然，读完本节会更好地理解各个参数对顶点的颜色是如何影响的。如果不允许光照，那么顶点的颜色只是当前颜色，如果允许光照，视点坐标系中会用到光照计算，下面对此加以介绍。

在下面的公式中，数学运算是对 RGB 分量分别进行的。举个例子，当 3 颜色值相加时，RGB 值是分别与相应的分量相加，再合成为最终结果 RGB 颜色的(R1+R2+R2，G1+G2+G3，B1+B2+B3)。当 3 个颜色值相乘时，计算结果形式为(R1R2R3，G1G2G3，B1B2B3)。(记住，某顶点颜色的 A 或 alpha 分量最终等于该点的材质散射特性值的 alpha 分量。)

顶点颜色的光照计算公式如下：

顶点颜色＝顶点发射光＋全局环境光×该点材质的环境光反射因子＋经过一定衰减的、各光源对该点贡献的环境反射光、漫射光、以及镜面反射光。

光照计算完毕后，该点的 RGBA 颜色值会被限制到[0，1]范围内。OpenGL 不会考虑物体对光线的阻挡，所以不会自动生成阴影效果。OpenGL 中物体的发射光不会照到其他物体上产生光照效果。要想产生那样的效果必须再多加光源才行。

8.6.1　材质发射光

这个最简单，就是给 GL_EMISSION 参数设置的 RGB 值。

8.6.2　放缩的全局环境光

放它们的全局环境光由全局环境光(GL_LIGHT_MODEL_AMBIENT 参数值)和材质的环境光反射特性(GL_AMBIENT 参数值)相乘得到。即，

```
ambientlight model * ambientmaterial
```

这 2 个参数的 RGB 值分别相乘，然后再合成为最终值：(R1R2，G1G2，B1B2)。

8.6.3　光源的贡献

每个光源都会对顶点的颜色有所贡献，而且这些贡献是可叠加的。计算每个光源贡献的公式如下：

```
contribution = attenuation factor * spotlight effect * (ambient term
+ diffuse term + specular term)
```

1. 衰减因子

衰减因子在 8.3.2 “位置与衰减” 一节中已经讨论过。

其中，d = 光源到顶点的距离，k_c = GL_CONSTANT_ATTENUATION，k_l = GL_LINEAR_ATTENUATION，k_q = GL_QUADRATIC_ATTENUATION，如果是方向光源，衰减因子为 1。

$$\text{衰减因子} = \frac{1}{k_c + k_l d + k_q d^2}$$

2. 聚光灯效果

聚光灯效果可以有 3 种评价值，这取决于光源是否是聚光灯以及顶点在光锥内部还是外部：

1，如果光源不是聚光灯(GL_SPOT_CUTOFF 等于 180)。

0，如果光源是聚光灯但是顶点位于光锥外部。

(max {v &μιδδοτ; d， 0 })*GL_SPOT_EXPONENT，其中，

v = (vx， vy， vz)是从聚光灯位置(GL_POSITION)指向顶点的单位向量。

d = (dx， dy， dz)是聚光灯方向，并假定是聚光灯光源且顶点位于光锥内部。

v 和 d 的点积代表这两个向量夹角的 cos 值，所以正对着聚光灯的点亮度最大，越靠边的点，cos 值越小，亮度也就越小。

为了判断一个点是否在光锥内部，OpenGL 计算(max {v &μιδδοτ; d， 0 })的值，如果该值比聚光灯发散角的 cos 值小，那么该点就在光锥外部，反之在内部。

3. 环境光分量

环境光分量只是光源的环境光分量和材质的环境光反射特性的乘积：

```
ambientlight *ambientmaterial
```

4. 漫射光分量

漫射光分量的计算要考虑光线是否垂直入射、入射光的漫射分量与材质的漫射特性：

(max { l &μιδδοτ; n ， 0 }) * diffuselight * diffusematerial，其中，

l = (lx， ly， lz)是从顶点指向光源位置(GL_POSITION)的单位向量。

n = (nx， ny， nz)是该顶点的单位法向量。

5. 镜面反射分量

镜面反射光分量的计算也要考虑光线是否垂直入射。如果 l &μιδδοτ; n 不大于 0，那么该点就没有镜面反射分量(如果小于 0，说明光源的光无法照到该计算的这一面来)。如果存在镜面反射分量的话，它由下面的因素决定：

- 该点的单位法向量 (nx， ny， nz)。
- 两个单位向量之和，它们是：由顶点指向光源位置的向量；由顶点指向视点的向量(假定 GL_LIGHT_MODEL_LOCAL_VIEWER 模式为真。如果为假，则第 2 个

向量由(0, 0, 1)代替)。最后的和向量要经过标准化，即除以自己的模变成单位向量，为 s = (sx, sy, sz)。

- 镜面反射指数 (GL_SHININESS)。
- 光源的镜面光分量(GL_SPECULAR)。
- 材质的镜面反射分量(GL_SPECULAR)。

在上面的定义下，OpenGL 是这样计算镜面反射分量的：

```
(max { s &μιδδοτ; n , 0} ) * shininess * specularlight * specularmaterial
```

如果 l &μιδδοτ; n = 0，上式为 0。

8.6.4 合成

由上面所定义的各个分量，整体的 RGBA 模式的光照计算为：

$$\text{顶点颜色} = \sum_{i=0}^{n-1}\left(\frac{1}{k_c + k_1 d + k_q d^2}\right)_i (spotlighteffect)_i$$

8.7 颜色索引模式中的光照

在颜色索引模式下，RGBA 形式的参数值或者没有意义或有其他的含义。因为在颜色索引模式下获得某种效果比较困难，所以应该 NYU 可能使用 RGBA 模式。实际上，在颜色索引模式下使用的 RGBA 形式的参数只是光源参数 GL_DIFFUSE 和 GL_SPECULAR 和材质参数 GL_SHININESS。这些参数(d_l, s_l)用来计算颜色索引模式下的漫射光和镜面反射光强 (d_{ci}, s_{ci})：

$d_{ci} = 0.30\ R(d_l) + 0.59\ G(d_l) + 0.11\ B(d_l)$

$s_{ci} = 0.30\ R(s_l) + 0.59\ G(s_l) + 0.11\ B(s_l)$

其中，R(x)，G(x)，和 B(x)代表颜色 x 的红、绿、蓝分量。权重系数 0.30，0.59 和 0.11 代表了视觉效果对各个颜色分量的权重——眼睛对绿色最敏感，而对蓝色最不敏感。在颜色索引模式下调用 glMaterial*()定义材质颜色时，要加入 GL_COLOR_INDEXES 参数如下：

```
GLfloat mat_colormap[] = { 16.0, 47.0, 3682.0 };
glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);
```

mat_colormap 的三个分量分别代表材质的环境光反射、漫射光反射、镜面光反射索引颜色值。换句话说，OpenGL 认为第一个索引值(本例中是 16.0)是纯的环境光反射色，第二个索引值(本例中是 47.0)是纯的漫射光反射色，第三个索引值(本例中是 3682.0)是纯的镜面反射光色。(默认状态下，上面 3 个值是 0.0, 1.0, 1.0。并注意 glColorMaterial()函数在颜色索引模式下无效)。

在画一幅场景时，OpenGL 利用这几个“纯”色之间的过渡值绘制物体的明暗效果。因此，必须建立这些纯色之间的过渡映射函数，在本例中是 16~47 之间，以及 47~3682 之间。常用的是光滑过渡，即线性函数，但也可以利用其他函数以达到不同的效果。下面的

例子给出了从黑色的环境光反射色到紫红色的漫射光反射色，再到白色的镜面光反射色的光滑过渡色的计算。

```
for (i = 0; i < 32; i++) {
auxSetOneColor (16 + i, 1.0 * (i/32.0), 0.0, 1.0 * (i/32.0));
auxSetOneColor (48 + i, 1.0, 1.0 * (i/32.0), 1.0);
}
```

其中，辅助函数 auxSetOneColor()有四个参数。第一个参数代表的索引值和后三个参数代表的 RGB 颜色对相关联。当 i=0 时，颜色索引 16 被赋与 RGB 值(0.0, 0.0, 0.0)，或黑色。随着 i 的增加，颜色过渡到索引值 47(i=31)，对应 RGB 值(1.0, 0.0, 1.0)，是纯紫红色。循环中的第 2 条语句在紫红和白色(索引值 3682)之间进行过渡。

8.8 发光小球碰撞实验

本节将模拟一些发光小球与周围墙面的碰撞实验，从而演示各种光照交互作用的效果。

```
#include <math.h>
#include <stdio.h>
#include <stdarg.h>
#include <GL/glaux.h>

#define UDIV 12
#define VDIV 12

#define WALLGRIDMAX 32
#define EYEZ 3.3f

#define TOTALBALLS 3

#define R 0
#define G 1
#define B 2

#define X 0
#define Y 1
#define Z 2
#define W 3

int wallgrid = 8;  /* 一面墙中方块的平方根，即网格单个方向的个数 */
float fatt = 1.0f;
```

```
int freeze = GL_FALSE;
int spin = GL_FALSE;
int objecton = GL_FALSE;
int normson = GL_FALSE;
int lighton[3] = {GL_TRUE, GL_TRUE, GL_TRUE};

struct {
  float p[3];
  float d[3];
  unsigned char color[3];
} balls[TOTALBALLS];

float ballobj[UDIV+1][VDIV+1][4];
float wallobj[WALLGRIDMAX+1][WALLGRIDMAX+1][4];
float wallnorms[WALLGRIDMAX+1][WALLGRIDMAX+1][3];
float wallnorm[3] = { 0.0, 0.0, -1.0 };

int orx, ory;

float ballscale;
float ballsize;

int DELTAX, DELTAY;

int lflag = 2;

float newpos[] = { 0.0f, 0.0f, 0.0f, 1.0f };

GLfloat light_Ka[] = { 0.3f, 0.3f, 0.3f, 1.0f };  /* 环境光颜色 */
GLfloat light_Ks[] = { 0.0f, 0.0f, 0.0f, 1.0f };  /* 镜面光颜色 */

GLfloat light0_Ka[]  = { 0.0f, 0.0f, 0.0f, 1.0f };  /* 环境光颜色 */
GLfloat light0_Kd[]  = { 1.0f, 0.1f, 0.1f, 1.0f };  /* 漫反射颜色 */
GLfloat light0_pos[] = { 0.0f, 0.0f, 0.0f, 1.0f };  /* 光源位置 */

GLfloat light1_Ka[]  = { 0.0f, 0.0f, 0.0f, 1.0f };  /* 环境光位置 */
GLfloat light1_Kd[]  = { 0.1f, 1.0f, 0.1f, 1.0f };  /* 漫反射位置 */
GLfloat light1_pos[] = { 0.0f, 0.0f, 0.0f, 1.0f };  /* 光源位置 */
```

```
GLfloat light2_Ka[]  = { 0.0f,  0.0f,  0.0f,  1.0f };   /* 环境光颜色 */
GLfloat light2_Kd[]  = { 0.1f,  0.1f,  1.0f,  1.0f }; /* 漫反射光颜色 */
GLfloat light2_pos[] = { 0.0f,  0.0f,  0.0f,  1.0f };   /* 光源位置 */

GLfloat attenuation[] = { 1.0f,  3.0f };

GLfloat plane_Ka[] = { 0.0f,  0.0f,  0.0f,  1.0f };   /* 环境光颜色 */
GLfloat plane_Kd[] = { 0.4f,  0.4f,  0.4f,  1.0f };  /* 漫反射光颜色 */
GLfloat plane_Ks[] = { 1.0f,  1.0f,  1.0f,  1.0f };   /* 镜面光颜色 */
GLfloat plane_Ke[] = { 0.0f,  0.0f,  0.0f,  1.0f };   /* 发射光颜色 */
GLfloat plane_Se   = 30.0f;                      /* 光亮度 */

GLfloat wall_Ka[] = { 0.1f,  0.1f,  0.1f,  1.0f };   /* 环境光颜色 */
GLfloat wall_Kd[] = { 0.8f,  0.8f,  0.8f,  1.0f };  /* 漫反射光颜色 */
GLfloat wall_Ks[] = { 1.0f,  1.0f,  1.0f,  1.0f };   /* 镜面光颜色 */
GLfloat wall_Ke[] = { 0.0f,  0.0f,  0.0f,  1.0f };   /* 发射光颜色 */
GLfloat wall_Se   = 20.0f;                      /* 光亮度 */

GLuint wall_material,  plane_material;  /* 材质显示列表 */

#define POLYGON   1
#define LINES   2
#define TRANSPERENT 3
#define DISPLAY   4
#define LMATERIAL 5

#define FASTMAGIC   0x5423

typedef struct fastobj {
   int npoints;
   int colors;
   int type;
   int material;
   int display;
   int ablend;
   int *data;
} fastobj;
```

```
fastobj *readfastobj();
fastobj   *obj = NULL;

float frand(void)
{
    return 2.0f*(rand()/32768.0f -0.5f);
}

void resetballs(void)
{
    register short i;

    balls[0].color[R] = 255;
    balls[0].color[G] = 64;
    balls[0].color[B] = 64;
    balls[1].color[R] = 64;
    balls[1].color[G] = 255;
    balls[1].color[B] = 64;
    balls[2].color[R] = 64;
    balls[2].color[G] = 64;
    balls[2].color[B] = 255;
    for (i = 0;  i < TOTALBALLS;  i++) {
        balls[i].p[0] = 0.0f;
        balls[i].p[1] = 0.0f;
        balls[i].p[2] = 0.0f;
        balls[i].d[0] = 0.1f*frand();
        balls[i].d[1] = 0.1f*frand();
        balls[i].d[2] = 0.1f*frand();
    }
}

void drawface(void)
{
    register int i, j;

    glNormal3fv(wallnorm);
    for (i=0;  i < wallgrid;  i++) {
    glBegin(GL_TRIANGLE_STRIP);
    for (j=0;  j <= wallgrid;  j++) {
```

```
        glVertex3fv(wallobj[i][j]);
        glVertex3fv(wallobj[i+1][j]);
    }
    glEnd();
    }
}

void drawnorms(void)
{
    register int i, j;

    glDisable(GL_LIGHTING);
    glColor3ub(255, 255, 0);
    for (i=0; i <= wallgrid; i++) {
    for (j=0; j <= wallgrid; j++) {
        glBegin(GL_LINES);
        glVertex3fv(wallobj[i][j]);
        glVertex3fv(wallnorms[i][j]);
        glEnd();
    }
    }
    glEnable(GL_LIGHTING);
}


void drawbox(void)
{
    glPushMatrix();

    glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
    drawface();
    if (normson) drawnorms();
    glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
    drawface();
    if (normson) drawnorms();
    glRotatef(90.0f, 0.0f, 1.0f, 0.0f);
    glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
    drawface();
    if (normson) drawnorms();
```

```
    glRotatef(180.0f, 1.0f, 0.0f, 0.0f);
    glPopMatrix();
}


void drawfastobj(fastobj *obj)
{
    register int *p, *end;
    register int npolys;

    p = obj->data;
    end = p + 8 * obj->npoints;

    if(obj->colors) {
    npolys = obj->npoints/4;
    p = obj->data;
    while(npolys..) {
        glBegin(GL_POLYGON);
        glColor3iv(p);
        glVertex3fv((float *)p+4);
        glColor3iv(p+8);
        glVertex3fv((float *)p+12);
        glColor3iv(p+16);
        glVertex3fv((float *)p+20);
        glColor3iv(p+24);
        glVertex3fv((float *)p+28);
        glEnd();
        p += 32;
    }
    } else {
    while ( p < end) {
        glBegin(GL_POLYGON);
        glNormal3fv((float *)p);
        glVertex3fv((float *)p+4);
        glNormal3fv((float *)p+8);
        glVertex3fv((float *)p+12);
        glNormal3fv((float *)p+16);
        glVertex3fv((float *)p+20);
        glNormal3fv((float *)p+24);
```

```
        glVertex3fv((float *)p+28);
        glEnd();
        p += 32;
    }
    }
}

void drawball(void)
{
    register int i, j;

    for (i=0; i < UDIV; i++) {
    for (j=0; j < VDIV; j++) {
        glBegin(GL_POLYGON);
        glVertex4fv( ballobj[i][j] );
        glVertex4fv( ballobj[i+1][j] );
        glVertex4fv( ballobj[i+1][j+1] );
        glVertex4fv( ballobj[i][j+1] );
        glEnd();
    }
    }
}

void  CALLBACK drawimage(void)
{
    register short i;
    static int start, end, last;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    for (i=0; i < TOTALBALLS; i++) {
        newpos[0] = balls[i].p[0];
        newpos[1] = balls[i].p[1];
        newpos[2] = balls[i].p[2];
        glLightfv(GL_LIGHT0 + i, GL_POSITION, newpos);
    }

    glCallList(wall_material);
    glEnable(GL_LIGHTING);
```

```
        drawbox();

        glEnable(GL_DEPTH_TEST);

        if (objecton)
        {
            glCallList(plane_material);
            glPushMatrix();
            glScalef(1.5f, 1.5f, 1.5f);
            glRotatef(180.0f, 0.0f, 0.0f, 1.0f);
            if (spin)
            {
                orx += 50;
                ory += 50;
            }
            glRotatef(orx/10.0f, 1.0f, 0.0f, 0.0f);
            glRotatef(ory/10.0f, 0.0f, 1.0f, 0.0f);
            drawfastobj(obj);
            glPopMatrix();
        }

        glDisable(GL_LIGHTING);

        for (i=0; i < TOTALBALLS; i++) {
            if (lighton[i])
            {
                glPushMatrix();
                glTranslatef(balls[i].p[0] , balls[i].p[1] , balls[i].
    p[2]);
                glColor3ubv(balls[i].color);
                drawball();
                glPopMatrix();
            }
        }

        glColor3f(1.0f, 1.0f, 1.0f);
        glPopMatrix();
        auxSwapBuffers();
    }
```

```
void initobjects(void)
{
    register float u, v, du, dv;
    register short i, j;

    du = 2.0f*3.1416f/UDIV;
    dv = 3.1416f/VDIV;

    u = 0.;
    for (i=0; i <= UDIV; i++) {
        v = 0.;
    for (j=0; j <= VDIV; j++) {
    ballobj[i][j][X] = ballsize*(float)cos(u)*(float)sin(v);
    ballobj[i][j][Y] = ballsize*(float)sin(u)*(float)sin(v);
            ballobj[i][j][Z] = ballsize*(float)cos(v);
            ballobj[i][j][W] = 1.0f;
            v += dv;
        }
        u += du;
    }

    for (i=0; i <= wallgrid; i++) {
        for (j=0; j <= wallgrid; j++) {
            wallobj[i][j][X] =-1.0f + 2.0f*i/wallgrid;
            wallobj[i][j][Y] =-1.0f + 2.0f*j/wallgrid;
            wallobj[i][j][Z] =1.0f;
            wallobj[i][j][W] =1.0f;
        }
    }

    for (i=0; i <= wallgrid; i++) {
        for (j=0; j <= wallgrid; j++) {
    wallnorms[i][j][X] = wallobj[i][j][X] + wallnorm[X]*0.1f;
    wallnorms[i][j][Y] = wallobj[i][j][Y] + wallnorm[Y]*0.1f;
    wallnorms[i][j][Z] = wallobj[i][j][Z] + wallnorm[Z]*0.1f;
        }
    }
}
```

```
void initialize(void)
{
    initobjects();
    srand(29716);

    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, light_Ka);

    plane_material = glGenLists(1);
    glNewList(plane_material, GL_COMPILE);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, plane_Ka);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, plane_Kd);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, plane_Ks);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, plane_Ke);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, plane_Se);
    glEndList();

    wall_material = glGenLists(1);
    glNewList(wall_material, GL_COMPILE);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, wall_Ka);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, wall_Kd);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, wall_Ks);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, wall_Ke);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, wall_Se);
    glEndList();

    glLightfv(GL_LIGHT0, GL_AMBIENT, light0_Ka);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_Kd);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, attenuation[0]);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, attenuation[1]);
    /* OpenGL 的 0 光与其他光有不同的镜面光属性 */
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_Ks);

    glLightfv(GL_LIGHT1, GL_AMBIENT, light1_Ka);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_Kd);
    glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, attenuation[0]);
    glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, attenuation[1]);
```

```
        glLightfv(GL_LIGHT2, GL_AMBIENT, light2_Ka);
        glLightfv(GL_LIGHT2, GL_DIFFUSE, light2_Kd);
        glLightf(GL_LIGHT2, GL_CONSTANT_ATTENUATION, attenuation[0]);
        glLightf(GL_LIGHT2, GL_LINEAR_ATTENUATION, attenuation[1]);
    }

    void calcball(void)
    {
        register short i, j;

        for (j=0; j < TOTALBALLS; j++) {
            for (i=0; i < 3; i++) {
                balls[j].p[i] += balls[j].d[i];
                if (fabs(balls[j].p[i]) > ballscale) {
                    balls[j].p[i] = (balls[j].p[i] > 0.0) ?
                    ballscale : -ballscale;
                    balls[j].d[i] = -balls[j].d[i];
                }
            }
        }
    }

void CALLBACK reshape(int width, int height)
{
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0f, (float)width/height, EYEZ-2.0f, EYEZ+2.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
    glTranslatef(0.0f, 0.25f, -EYEZ);
}

void CALLBACK idle(void)
{
    if (!freeze)
        calcball();
    drawimage();
}
```

```
void main()
{
    auxInitPosition(0, 0, 512, 512);
    initialize();
    auxInitDisplayMode(AUX_RGBA | AUX_DEPTH | AUX_DOUBLE);
    auxInitWindow("Bounce");
    ballsize = 0.04f;
    ballscale = 1.0f -ballsize;

    initialize();
    resetballs();
    /* 在盒子中使用局部光照模型 */
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHT2);
    auxIdleFunc(idle);
    auxReshapeFunc(reshape);
    auxMainLoop(drawimage);
}
```

程序运行结果如图 8.6 所示。

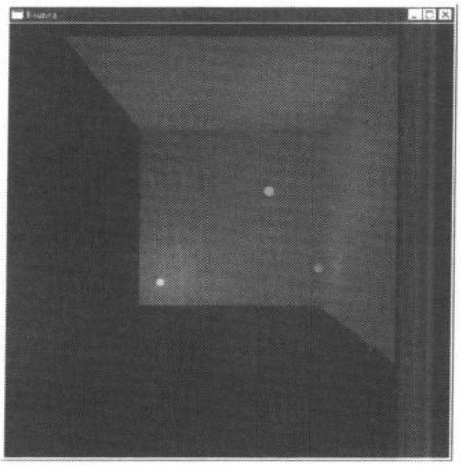

图8.6　发光小球碰撞实验

第 9 章 读取三维图形数据

在三维图形程序中，经常要绘制许多复杂的图形，如一个非常精细的动物模型或一个逼真的地形场景等，单纯利用 OpenGL 的实例库提供的基本几何体构造着实困难。而且也不可能一次性地在内存中编写绘图语句。必须考虑一些合适的三维数据存储结构，使之可以将复杂的三维模型保存下来。最常用的保存三维模型的方法是用多边形逼近的方法，即用许多小多边形拼出模型的外观，文件中保存这些多边形的信息。由于 OpenGL 中提供了最基本的由多边形构造三维模型的方法，因此从三维图形数据文件中读取模型数据，在 OpenGL 中进行绘制就非常容易了。

9.1 三维图形数据文件格式简介

存储三维图形数据的文件格式有多种，其中比较常见的有 Wavefront(.obj)格式、3DS 格式、用于在 Internet 上传输三维图形的 WRL 格式如 AutoCad(.dxf)格式等。下面以最为常见的两种三维图形数据文件格式为例，具体说明三维图形数据格式的存储方式。

对于文件格式的详细描述可以帮助更好地理解三维信息是如何组织的，从而更好地掌握有关三维图形的知识，对于编写了建模和造型相关的程序也很有帮助。

9.1.1 Wavefront 模型简介

Wavefront 的.obj 模型文件结构非常简单(当然 Wavefront 同时也提供了二进制文件的存储方式.mod，在此介绍)。它是由一些文本行组成的，每一行文本都是由一个关键字开头，紧跟着后面是该关键字所表示含义的具体参数，例如，以 v 开头表示该行将描述一个顶点信息，那么后面一定将跟着 3 个 double 型字符串，分别用于描述该顶点的三维坐标。

.obj 文件不仅可以定义多边形模型，也可以定义自由曲面模型。多边形模型就是可以用点、线、面等基本元素构造的模型，而自由曲面模型是指使用参数曲面造型定义的模型。表 9.1 中给出了.obj 文件中最常用的一些关键字及其所表示的含义。

表9.1 Wavefront关键字取值及其含义

关键字	含 义
v	顶点几何坐标
vt	顶点纹理坐标
vn	顶点法向量
vp	参数空间顶点

续表9.1

关键字	含　义
cstype	有理或无理曲面造型
deg	角度
bmat	基本矩阵
step	步长
p	点
l	线
f	面
curv	曲面
g	组名称
s	平滑组
mg	合并组
o	对象名称
usemtl	材质名称
mtllib	材质库

下面按每种关键字的复杂程度从简单到复杂，分别说明它们的语法定义。

顶点的语法定义如下：

```
v x y z w
```

其中 w 是一个齐次坐标，一般情况下可以默认，默认值为 1.0。x，y，z 分别表示顶点位置的三维坐标，它们都取 float 型值。

注意：建议 w 值最好设置为正值，0 或负值可能会在曲线中引入一个未定义顶点。

法向量的语法定义如下：

```
vn i j k
```

可用于指定法向量的 3 个分量。vn 主要影响绘制几何场景时的平滑明暗处理等效果。对于多边形模型，一般指定顶点法向量，而不是面法向量。面法向量在几个顶点之间插值。

纹理坐标的语法定义如下：

```
vt u v w
```

其中一维纹理坐标仅使用 u 坐标，二维纹理使用 u，v 两个坐标，三维纹理则使用所有 3 个坐标。其中 u 代表横向纹理，v 代表垂直方向纹理，w 是一个可选项，可用于表示纹理的深度，默认值为 0。

一个多边形的语法定义如下：

```
f v/vt/vn v/vt/vn v/vt/vn v/vt/vn ...
```

其中省略号部分表示，可以定义任意多边的多边形。每一个元素之间需用“/”符号隔开，例如，可以用以下语句定义一个四边形：

```
f 1/1/1 2/2/2 3/3/3 4/4/4
```

如果只有顶点坐标和法向量坐标(不含纹理坐标)，则应该使用“//”将坐标隔开。例如，

只用顶点坐标和法向量坐标定义一个四边形：

```
f 1//1 2//2 3//3 4//4
```

以下列出几种非常简单而常用的几何模型的定义方法：

- 正方形

例定义了一个四边形，其中四边形的每一边的长度均为 2，且为前向面。

```
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
```

- 立方体

以下定义了一个立方体，其每一边的边长均为 2，每个顶点由 3 条边共享。

```
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
f 8 7 6 5
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
```

注意：与通常 C 语言编程时使用下标的情况不同，在 Wavefront 文件中，是从 1 号开始编号的。

9.1.2　3DS 文件格式

为了深入地了解 3DS 文件的格式，帮助用户理解上述实例中的各个函数的用法，将对 3DS 文件的格式作详尽的描述。这里描述适合于 3DS 3.0 以上格式。

分以下 3 部分内容：3DS 文件格式简介；3D 编辑程序块；关键帧块。

1.　3DS 文件格式简介

3DS 文件由许多块组成，每个块首先描述其信息类别，即该块是如何组成的。块的信息类别用 ID 来标识，块还包含了下一个块的相对位置信息。因此，即使不理解一个块的含义，也可以很容易地跳过它，因为该块中指出了下一个块相对于该块起始位置的偏移字节

数。与许多文件格式一样，3DS 二进制文件中的数据也是按低位在前、高位在后的方式组织的，例如，2 个十六进制字节，4A 5C 组成的整型数，表明 5C 是高位字节，4A 是低位字节；对于长整型数，如：4A 5C 3B 8F 表明 5C4A 是低位字，而 8F3B 是高位字。

下面描述块的具体定义。块的前两项信息分别是：块的 ID 和块的长度(也即下一个块相对于该块的字节偏移量)，块的 ID 是一个整型数，而块的长度是一个长整型数。每个块实际上是一个层次结构，不同类型的块，其层次结构也不相同。3DS 文件中有一个基本块，其 ID 是 4D4D，每一个 3DS 文件的开头都是由这样一个块构成。基本块内的块称为主块。

为了对块的层次结构有一个初步的认识，下面给出一个图表来说明不同类型(ID)的块及其各自在文件中的位置。如图 9.1 所示给每一个块都起了一个名字，这样更便于理解或直接转换为源程序。

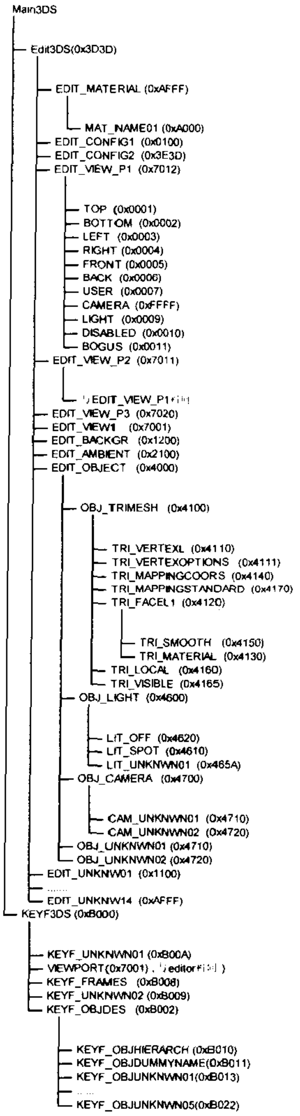

图9.1　3DS文件结构图

颜色块是文件中自始至终都能见到的一种块类型，颜色块总共有 3 种类型，它们是 COL_RGB，COL_TRU 和 COL_UNK。

2. 3D编辑程序块

最开始出现的主块是基本块，它包含了整个文件。因此这个块的大小就是文件的大小减去主块的大小。还有 2 种主块：3D 编辑程序块和关键帧块，前者的 ID 是 3D3D， 后者的 ID 是 B000。3D 编辑程序块表明编辑程序数据开始，也就是说物体的形体数据定义从此处开始。关键帧块表明下面开始定义关键帧信息。

主块后面又跟着其他块，它可以是在这个主块中允许的任何一种块类型，称之为该主块的子块。

- 3D(编辑程序块)的子块

ID	含义
0100	配置的一部分
1100	未知
1200	背景色
1201	未知
1300	未知
1400	未知
1420	未知
1450	未知
1500	未知
2100	环境颜色块
2200	雾
2201	雾
2210	雾
2300	未知
3000	未知
3D3E	编辑程序配置主块
4000	一个物体的定义
AFFF	材质列表开始

- AFFF(材质列表块)的子块

A000　材质名称　　用一个零结尾的字符串定义

- 3D3E(编辑程序配置块)的子块

ID	含义
7001	视口指示器
7011	视口定义(类型 2)
7012	视口定义(类型 1)
7020	视口定义(类型 3)

3D3E 块中包含了许多冗余数据，其中较重要的块是 7020 块，这个块定义编辑程序中 4 个活动的视口。假设在编辑程序布局中使用了 4 个视口，编辑程序配置中包含 5 个 7020 视口定义块和五个 7011 视口定义块。但事实上只有开始的四个 7020 视口定义块对用户的

视口的外观有影响，其他的视口中只包含一些附加信息。该块的第 6，7 字节表明视图的类型，合法的 ID 及其对应的视图如下所示：

ID	含义
0001	顶视图
0002	底视图
0003	左视图
0004	右视图
0005	前视图
0006	后视图
0007	用户视图
FFFF	相机视图
0009	光源视图
0010	无效

- 4000 (物体描述块)的子块

第一项是物体名称，它是一个零结尾串，这里所指的物体可以是一个相机或一个光源，也可以是物体的形体

ID	含义
4010	未知
4012	阴影块
4100	三角形列表块
4600	光源块
4700	相机块

- 4100 (三角形列表)的子块

ID	含义
4110	顶点列表
4111	顶点选项
4120	面列表
4130	面材质
4140	纹理映射坐标
4150	面平滑组
4160	平移矩阵
4165	物体可见性
4170	标准映射

- 4110(顶点列表块)的子块

起始位置	结束位置	大小	数据类型	名称
0	1	2	无符号整数	物体的顶点数目(VertexNum)
2	5	4	浮点型	X 坐标值
6	9	4	浮点型	Y 坐标值
10	13	4	浮点型	Z 坐标值

从第 2 个字节到第 13 个字节定义了一个顶点的坐标，重复这项定义 VertexNum 次，便得到了所有顶点的坐标。

- 4111(顶点选项块)的子块

该子块由一些整型数组成，第一个整型数表明顶点个数，然后对每个顶点用一个整型数表示一些位信息，其中，0~7，11~12 位影响物体的可见性，8~10 位是随机信息，13~15 位表明该顶点是否在某个选择集中被选中。顶点选项块不是很重要，即使将其删除，3DS 也能正确地将物体装入。

- 4120 面列表块

起始位置	结束位置	大小	数据类型	名称
0	1	2	无符号整数	物体中的三角形数(PolyNum)
2	3	2	无符号整数	顶点 A 的序号
4	5	2	无符号整数	顶点 B 的序号
6	7	2	无符号整数	顶点 C 的序号
8	9	2	无符号整数	面信息

重复 2~9 定义 PolyNum 次，就给出了所有的三角形。2~7 所给出的 3 个整型数是三角形面的 3 个顶点序号，序号为 0 的顶点表示顶点列表中定义的第一个顶点。顶点的顺序影响着面的法向量方向。一般情况下三角形应按逆时针方向定义，但有些 3DS 文件使用顺时针方向，这时就需要在程序中将其调整过来。面信息是一个整型数，其中前三个二进制位给出了三角形每条边的顺序，可以根据它们判别三角形是以逆时针还是顺时针顺序给出的。这 3 位数字要么全是 0，要么全是 1，若这三位全为 1 ，其二进制方式为 111，意味着三角形的三顶点的正确顺序应是 A→C→B，即 A→B→C 给出的是顺时针方向。反之，这 3 位全是 0，则意味着 A→B→C 给出的就是逆时针方向。下面给出面信息中每一位的具体含义：

bit 0	AC 边顺序，1：A→C　0：C→A
bit 1	BC 边顺序　1：B→C　0：C→B
bit 2	AB 边顺序　1：A→B　0：B→A
bit 3	纹理映射(如果有)
bit 4~8	保留(0)
bit 9~10	随机数
bit 11~12	保留(0)
bit 13	面在第 3 选择集中被选中
bit 14	面在第 3 选择集中被选中
bit 15	面在第 3 选择集中被选中

- 4130 面材质块

材质块给出了物体中使用的每一种材质，但并不是每个物体都有材质块，只使用默认材质的物体就没有材质块。每一个 4130 面材质块以一个 0 结尾的材质名称字符串开始，接着跟一个由数字表示的与该材质相关的面的数目，然后就是一个一个的面。0000 表示 4120 面列表中的第一个面。

- 4140 纹理映射坐标

前 2 个字节表示定义的顶点数，然后为每个顶点定义 2 个浮点型的纹理映射坐标。也

就是说，如果一个顶点的纹理映射在纹理平面的中心，那么其映射坐标即为(0.5，0.5)。

- 4150 面平滑组块

块的大小为面的个数×4 个字节，即每个基本数据是一个长整型数字，第 n 个数字表示该面是否属于第 n 个平滑组。

- 4160 局部坐标轴块

局部坐标轴信息，开始的 3 个浮点数定义了物体的局部坐标轴在绝对坐标系内的 X，Y，Z 坐标。最后的 3 个浮点数是物体的局部中心。

- 4170 标准映射

前 2 个字节表明映射的类型，其中，0 表明平面映射或指定映射，当指定映射时，与这个块中的信息就无关了，1 表明圆柱映射，2 表明球映射。接着用 21 个浮点数来描述这个映射。

- 光源块

ID	含义
0010	RGB 颜色块
0011	24 位颜色
4610	点光源
4620	光源被关闭

起始位置	结束位置	大小	数据类型	名称
0	3	4	浮点型	光源位置 X 坐标
4	7	4	浮点型	光源位置 Y 坐标
8	11	4	浮点型	光源位置 Z 坐标

- 4610　点光源块

起始位置	结束位置	大小	数据类型	名称
0	3	4	浮点型	目标位置 X 坐标
4	7	4	浮点型	目标位置 Y 坐标
8	11	4	浮点型	目标位置 Z 坐标
12	15	4	浮点型	热点
16	19	4	浮点型	发散

- 0010　RGB 颜色块

起始位置	结束位置	大小	数据类型	名称
0	3	4	浮点型	红
4	7	4	浮点型	绿
8	11	4	浮点型	蓝

- 0011　24 位 RGB 颜色块

起始位置	结束位置	大小	数据类型	名称
0	1	1	浮点型	红
1	1	1	浮点型	绿
2	2	1	浮点型	蓝

- 相机块

起始位置	结束位置	大小	数据类型	名称
0	3	4	浮点型	相机位置 X 坐标
4	7	4	浮点型	相机位置 Y 坐标
8	11	4	浮点型	相机位置 Z 坐标
12	15	4	浮点型	相机目标 X 坐标
16	19	4	浮点型	相机目标 Y 坐标
20	23	4	浮点型	相机目标 Z 坐标
24	27	4	浮点型	相机倾斜（旋转角）
28	31	4	浮点型	相机镜头

3. 关键帧块

关键帧块的 ID 及其含义如下所示。

ID	含义
B00A	未知
B008	帧信息块
B009	未知
B002	物体信息开始块

- B008 帧信息块

起始位置	结束位置	大小	数据类型	名称
0	3	4	无符号长整数	开始帧
4	7	4	无符号长整数	结束帧

- B002 物体信息开始块

物体信息开始块的子块 ID 及其含义如下所示。

子块 ID	含义
B010	名称与层次结构描述
B011	哑元物体命名
B013	未知
B014	未知
B015	未知
B020	物体中心点
B021	未知
B022	未知

- B010 名称与层次结构描述块

起始位置	结束位置	大小	数据类型	名称
0	未确定	?	0 结尾字符串	物体名称
未确定	未确定	2	unsigned int	未知
未确定	未确定	2	unsigned int	未知
未确定	未确定	2	unsigned int	物体的层次结构

物体的层次结构并不复杂，场景中给予每个物体一个数字以标识其在场景树中的顺序。

相应地 3DS 文件中也用同样的方法标识了物体在场景树中的位置。作为根物体给予了数字-1(FFFF)作为其数字标识。当读取文件的时候，就会得到一系列的物体数字标识。如果当前数字标识比前一个大，那么当前物体是前一个物体的子物体；反之，当数字标识比前一个数字标识小，则又回到上一层结构。

9.1.3 各种 3D 数据模型相互转换

在编写程序时不希望将精力花费在一些琐碎事情上面，由于三维模型的类型比较多，无法为每一种都编写读取数据的程序，当需要使用某种格式的模型文件时，希望有一种工具能够将其转换成一种可以读取的格式。

对于三维图形数据模型，有很多软件提供了读取其他格式的能力，这些程序通常很大，因为它们不仅提供格式转化的功能，通常还包括建模和绘制等功能。对于一般编写图形程序的人员来说，实在是大材小用，这就需要有一些专门实现格式转换的小巧的软件。

这样的软件不多，作者喜欢使用的一个工具是 convert，它可以在 3ds，obj，dxf，wrl 等格式之间进行自由的转换，它甚至还能将三维图形数据文件转化成用它们进行绘制的 OpenGL 的 C 语言源程序。美中不足的是，对于 3ds、dxf 等模型存储的数据，它只能保存和转换几何信息，材质等信息则在转换时全部丢失。

9.2 在 OpenGL 中读取和操作 3DS 模型文件

3D.Studio 是 Autodesk 公司在微机上开发的一套用于制作三维动画的应用软件，它的功能非常强大，至今许多电视广告的前期制作仍在使用它。许多用户可能曾经使用过它，并用它制作出许多美观而又有趣的计算机动画，也一定为它的强大功能所折服，对于一个程序员，都希望能够在自己的程序中方便自如地操作和管理自己用辛勤的汗水制作出的精美作品。

本实例就是满足这些用户需要的。实例详尽而又清晰地讲解了如何读取一个 3DS 文件，并用 OpenGL 对其描述的场景进行绘制。

实例共分为 3 个部分，第一部分讲解如何读取 3DS 文件到自定义的图形数据结构中，第二部分讲解如何对上述数据结构进行各种几何运算，形成可以用于绘制的数据，第三部分给出程序中用到的各种矩阵运算函数。

用户可以对该程序稍加修改或补充，例如，为其加入各种图形操作功能，或者根据它的数据结构将它转换为其他类型的三维图形文件，甚至还可以用它来制作自己的动画等，这样就可以制作出比较专业的三维图形软件。另外，由于 3DS 是一种非常普遍的数据格式，以 3DS 格式保存的三维图形文件非常丰富。种类繁多的各种三维图形素材光盘以及内容丰富的网上图形站点，都有非常丰富的 3DS 文件库。微机上的 3D Studio 软件制作起来也很方便。因此学会读取和操作 3DS 文件对于建立比较实用的三维图形应用软件显得非常重要。

程序中用到的数据结构可能不太容易理解，因此这里先作简要介绍。程序中共用到 3 种主要的数据结构：顶点集(Vertices)、多边形列表(PolyList)和相关三角形列表

(CommonTriangleList)。顶点集(Vertices)是顶点及其法向量的列表，多边形中的顶点将用在下标表示，OpenGL 将这个数组视为一个顶点数组(vertex array)。这里存储的法向量是每一个顶点的法向量，而不是多边形的法向量。多边形列表(PolyList) 包括一系列的材质，一个材质由这样一组属性构成：漫反射颜色、光照度和纹理映射等。与每一种材质相对应是一组三角形，每个三角形由一个三元整数组表示，它们分别是三角形每个顶点在顶点集中的下标。例如：

```
Vertices[PolyList[2].TriangleIndexList[3].p0]
Vertices[PolyList[2].TriangleIndexList[3].p1]
Vertices[PolyList[2].TriangleIndexList[3].p2]
```

分别表示第三组材质的第 4 个三角形的 3 个顶点，当然每个顶点(p0，p1，p2)都有相关的 GLfloat 型数据代表其 X，Y，Z 值。相关三角形列表(CommonTriangleList)与顶点集(Vertices)具有相同的大小，它是一个指向 PolyList[].TriangleIndexList 中每一个包含该顶点的三角形的链表，例如：

```
for (ptr = CommonTriangleList[5]; ptr; ptr=ptr.>next)
   printf("%d %d %d\n",
         ptr->triangle->p0,
         ptr->triangle->p1,
         ptr->triangle->p2);
```

将会遍历一个链表，显示与第 6 个顶点相关的所有 3 角形的三个顶点的下标，这里的第 6 个顶点也就是 Vertices[5]。CommonTriangleList 被用来建立每个顶点的法向量，有时为了生成硬边界，需要为同一个顶点指定两个以上法向量，这就需要判别一个顶点周围的各个面是否形成硬边界，以及如何给顶点指定法向量。

读取 3DS 文件到自定义的图形数据结构中

1. 一组文件操作函数

- 从文件中读入 len 个字节的数据到 dest 指针指向的内存中。

```
void dread(void *dest, int len)
{
  if(fread(dest, len, 1, InFile) != 1)
    longjmp(EnvState, 1);
}
```

- 将文件指针移到指定的位置。

```
void dsetpos(dword pos)
{
  if(fseek(InFile, (long)pos, SEEK_SET) != 0)
    longjmp(EnvState, 1);
}
```

- 找到文件当前位置。

```
dword dgetpos(void)
{
  long pos;
  if((pos=ftell(InFile)) ==-1L)
    longjmp(EnvState, 1);
  return (dword) pos;
}
```

2. 内存申请和释放

- 安全地分配一定大小的内存。

```
void *getmem(int size)
{
  void *mem;

  if((mem=malloc(size))==0)
    longjmp(EnvState, 1);
  return mem;
}
```

- 从内存中将 H3dsScene 删除。

```
void HFree3dsScene(H3dsScene * scene)
{
  int n, m;

  if(scene){
     if(scene->meshobjlist){
        for(n=scene->meshobjs-1; n>=0; n-) {
           H3dsMeshObj *mobj = &scene->meshobjlist[n];
           if(mobj->maplist)
              free(mobj->maplist);
           if(mobj->facelist)
              free(mobj->facelist);
           if(mobj->vertlist)
              free(mobj->vertlist);
           if(mobj->bindings) {
              for (m=0; m<mobj->NumBindings; m++) {
           if (mobj->bindings[m].faces){
           free (mobj->bindings[m].faces);
        }
```

```
        }
    free (mobj->bindings);
          }
        }
        free(scene.>meshobjlist);
      }
      if(scene.>material) free (scene.>material);
         free(scene);
      }
    }
```

- 扩大 scene 数组，以装入新的形体对象。

```
H3dsMeshObj *EnlargeMeshArray(void) {
  void *mem;
  H3dsMeshObj *obj;

  if((mem = realloc(Scene->meshobjlist,
             sizeof(H3dsMeshObj)*(Scene->meshobjs+1))) == 0) {
    printf("Error reallocating mem\n");
    longjmp(EnvState, 1);
  }
  Scene->meshobjlist=(H3dsMeshObj *) mem;
  obj=&Scene->meshobjlist[Scene->meshobjs++];
  memset(obj, 0, sizeof(H3dsMeshObj));
  return obj;
}
```

- 扩大 scene 数组，以装入新的材质。

```
H3dsMat *EnlargeMatObjArray(void) {
  void *mem;
  H3dsMat *obj;

  if((mem = realloc(Scene->material,
             sizeof(H3dsMat)*(Scene->NumMaterials+1))) == 0) {
    longjmp(EnvState, 1);
  }
  Scene->material=(H3dsMat *) mem;
  obj = &Scene->material[Scene->NumMaterials++];
  memset(obj, 0, sizeof(H3dsMat));
  return obj;
}
```

● 扩大 scene 数组，以装入另一个捆绑的列表。

```
H3dsMatList *EnlargeBindingsArray(H3dsMeshObj *meshobj) {
void *mem;
H3dsMatList *obj;
if((mem = realloc(meshobj->bindings,
sizeof(H3dsMatList)*(meshobj->NumBindings+1))) == 0) {
   printf("Error reallocating mem\n");
   longjmp(EnvState, 1);
  }
  meshobj->bindings = (H3dsMatList *)mem;
  obj = &meshobj->bindings[meshobj->NumBindings++];
  memset(obj, 0, sizeof(H3dsMatList));
  return obj;
}
```

● 释放为模型分配的所有内存。

为模型建立的全局内存有顶点列表、三角形列表以及顶点相关三角形列表，这里将它们的内存一一释放。

```
void FreeData() {
 unsigned int i;
 TRIANGLENODE *TempPtr, *LList;

 free(Vertices);
 for (i=0; i<NumPolyLists; i++) {
  free (PolyList[i].TriangleIndexList);
  free (PolyList[i].CrunchedTriangleIndexList);
 }
 for (i=0; i<NumVertex; i++) {
  LList = CommonTriangleList[i];
  while (LList) {
   TempPtr = LList;
   LList = LList->next;
   free (TempPtr);
  }
 }
 free (CommonTriangleList);
}
```

3. 将3DS文件读到相应的数据结构中

● 读入当前形体对象的顶点列表。

3DS 文件中顶点的定义比较简单，它由一个表示顶点数目的整型数开始，接着是相应数目的顶点定义信息，每个顶点由 3 个浮点数分别表示其 3 个坐标。

```
void ReadVertList(H3dsMeshObj *meshobj)
{
  word nv;
  int n;

  dread(&nv, sizeof(nv));
  meshobj->NumVerts=nv;
  meshobj->vertlist=(H3dsVert *) getmem(sizeof(H3dsVert)*nv);
  for(n=0; n<nv; n++)
  {
    dread(&meshobj->vertlist[n].x, sizeof(float32));
    dread(&meshobj->vertlist[n].y, sizeof(float32));
    dread(&meshobj->vertlist[n].z, sizeof(float32));
  }
}
```

- 读入当前形体对象的面列表。

由于 3DS 中定义的每一个面都是三角形，因此，该函数从文件中读入的面列表就是三角形列表，每个三角形由它的 3 个顶点的序号以及一个面信息(标识)组成。文件中的第一项是三角形的数目，其后跟相应数目的三角形定义数据(包括 3 个顶点的序号和标识顶点顺序的面信息)，面信息中包含 3 个顶点的顺序。可以很容易地写出如下程序：

```
word ReadFaceList(H3dsMeshObj *meshobj)
{
  word nv;
  word swap;
  int n;

  // 先读出 3DS 文件中面的个数
  dread(&nv, sizeof(nv));
  meshobj->NumFaces=nv;
  // 根据面的个数申请足够的内存
  meshobj->facelist=(H3dsFace *) getmem(sizeof(H3dsFace)*nv);
  for(n=0; n<nv; n++)
  {
    dread(&meshobj->facelist[n].p0, sizeof(word));
    dread(&meshobj->facelist[n].p1, sizeof(word));
    dread(&meshobj->facelist[n].p2, sizeof(word));
```

```
        dread(&meshobj->facelist[n].flags, sizeof(word));

    // 面信息为 0x07 表示三角形是按顺时针方向定义的，而在 OpenGL 中要求多边形
    // 按逆时针方向给出，故需要将后面 2 个顶点的顺序交换
        if (meshobj->facelist[n].flags == 0x07) {
          swap = meshobj->facelist[n].p1;
          meshobj->facelist[n].p1 = meshobj->facelist[n].p2;
          meshobj->facelist[n].p2 = swap;
        }
      }
      return (nv);
    }
```

- 读入当前形体对象的纹理映射坐标。

与上面的顶点列表和面列表类似，纹理坐标列表也是由一个表示顶点数目的整型数开始，接着跟一组纹理坐标构成。每一组纹理坐标由 u，v 两个浮点值定义。

```
void ReadMapList(H3dsMeshObj *meshobj)
{
  word nv;
  int n;
  dread(&nv, sizeof(nv));
  meshobj->NumMaps=nv;
  //  为纹理坐标申请足够的内存
  meshobj->maplist=(H3dsMap *) getmem(sizeof(H3dsMap)*nv);
  for(n=0; n<nv; n++)
  {
    dread(&meshobj->maplist[n].u, sizeof(float32));
    dread(&meshobj->maplist[n].v, sizeof(float32));
  }
}
```

- 读入当前形体对象的平移矩阵。

形体对象的平移矩阵是 12 个浮点数组成的数组，表示一个 3×4 的矩阵。

```
void ReadTraMatrix(H3dsMeshObj *meshobj)
{
  int i;
  for (i=0; i<12; i++)
  dread(&meshobj->TraMatrix[i], sizeof(float32));
  meshobj->matrix=1;
}
```

- 为多边形指定块中的材质。

  在 3DS 中是根据名称索引材质的，文件中给出了每种名称的材质对应的多边形列表，以表明哪些多边形使用了该材质。每种材质首先给出的是它的名称，接着是与该材质相关的多边形个数，然后再给出相应数目的多边形在多边形列表中的下标。

```
void ReadFaceMaterial(H3dsMeshObj *meshobj)
{
  word nf;
  word face;
  char name[50];
  H3dsMatList *binding;
  int n;
  binding = EnlargeBindingsArray(meshobj);
  //  首先是材质的名称
  n=0;
  do {
    dread(&name[n++], 1);
  } while(name[n.1]!='\0' && n<sizeof(name));
  name[n.1]='\0';
  strcpy(binding->name, name);
  //  然后是多边形的个数
  dread(&nf, sizeof(nf));
  binding->NumFaces = (int)nf;
  binding->faces = (word *)(malloc(sizeof(face)*nf));
  // 将材质同用到该材质的多边形捆绑到一起
  for (n=0; n<nf; n++) {
    dread(&face, sizeof(face));
    binding->faces[n] = face;
  }
}
```

- 读入默认材质。

  为所有 nf 个多边形建立其默认材质 WG_DEFAULT。

```
void ReadDefaultMaterial(H3dsMeshObj *meshobj, word nf)
{
  word face=0;
  H3dsMatList *binding;
  int n;
  binding = EnlargeBindingsArray(meshobj);
  meshobj->NumBindings = 1;
```

```
    /* 首先是材质的名字 */
    strcpy(binding->name, "WG_DEFAULT");
    /* 然后是多边形的个数 */
    binding->NumFaces = (int)nf;
    binding->faces = (word *)(malloc(sizeof(face)*nf));
    /* 将多边形与材质捆绑到一起 */
    for (n=0; n<nf; n++) {
      binding->faces[n] = n;
    }
  }
```

- 读入形体的数据结构。

  对形体的描述包括：用三维坐标定义的顶点列表、三角形列表以及为每个面指定的材质等。

```
void ReadTriMeshBlocks(dword p, char *name)
{
  word id;
  word fl;
  dword len, pc;
  H3dsMeshObj *meshobj;
  int found_facemat;

  meshobj = EnlargeMeshArray();

  found_facemat = 0;
  strcpy(meshobj->name, name);
  while((pc=dgetpos()) < p) {
    dread(&id, sizeof(id));
    dread(&len, sizeof(len));
    switch((int)id) {
      // 若是顶点列表块
      case CHUNK_VERTLIST:
          // 读入顶点列表信息
           ReadVertList(meshobj);
           break;
      // 若是三角形列表块
      case CHUNK_FACELIST:
          读入三角形列表信息
           fl = ReadFaceList(meshobj);
           break;
```

```
            //  若是纹理映射列表块
            case CHUNK_MAPLIST:
                //  读入纹理映射列表
                 ReadMapList(meshobj);
                 break;
            //  若是平移矩阵块
            case CHUNK_TRMATRIX:
                //  读入平移矩阵
                 ReadTraMatrix(meshobj);
                 break;
            //  若是材质对应的面列表块
            case CHUNK_FACEMAT:
                //  读入材质对应的面列表
                 ReadFaceMaterial(meshobj);
                 found_facemat = 1;
                 break;
            default:
                //  其他块，简单地跳过这个块
                 dsetpos(pc+len);
        }
    }
    //  若没有指定面所对应的材质
    if (!found_facemat) {
      //  则使用默认材质
      defaultmaterialwarning = 1;
      ReadDefaultMaterial(meshobj, fl);
    }
}
```

- 读入物体块。

  物体块中包括相机、形体或光照等信息，这里只处理形体信息，而简单地跳过其他类型的信息。

```
void ReadObjBlocks(dword p)
{
  word id;
  dword len, pc;
  char name[16];
  // 第一项总是物体的名称
  int n=0;
  do {
```

```
      dread(&name[n++], 1);
    } while(name[n.1]!='\0' && n<sizeof(name));
    name[n.1]='\0';
    while((pc=dgetpos()) < p) {
      dread(&id, sizeof(id));
      dread(&len, sizeof(len));
      switch((int)id) {
        case CHUNK_TRIMESH:
             ReadTriMeshBlocks(pc+len, name);
             break;
        default: dsetpos(pc+len);
      }
    }
  }
```

- 读入材质的名称。

```
void ReadMatChunk(dword p, H3dsMat *mat) {
  int n;
  char name[16];
  n = 0;
  // 材质的名称就是一个 0 结尾串
  do {
    dread(&name[n++], 1);
  } while(name[n.1]!='\0' && n<sizeof(name));
  name[n.1]='\0';
  strcpy(mat.>name, name);
}
```

- 读入一个 RGB 三元组。

  对颜色的存储，3DS 有两种 RGB 方式，一种是用浮点型 RGB 值表示，另一种是用字节型的 RGB 值表示。这里给出读取这两种形式的颜色三元组的程序：

- 读取浮点型的颜色三元组的函数：

```
void ReadRGBFloat(dword p, H3dsRGBColor *rgb) {
  dread(&rgb->red, sizeof(float32));
  dread(&rgb->green, sizeof(float32));
  dread(&rgb->blue, sizeof(float32));
}
```

- 读取字节型颜色三元组的函数，将其转换成浮点型存储。

```
void ReadRGBByte(dword p, H3dsRGBColor *rgb) {
  unsigned char red, green, blue;
  dread(&red, sizeof(red));
```

```
        dread(&green, sizeof(green));
        dread(&blue, sizeof(blue));
        rgb->red = (float)(red)/256.0f;
        rgb->green = (float)(green)/256.0f;
        rgb->blue = (float)(blue)/256.0f;
    }
```

● 读入一个阴影块。

```
void ReadShadingChunk(dword p, H3dsRGBColor *rgb) {
  word id;
  dword len, pc;
  while((pc=dgetpos()) < p) {
    dread(&id, sizeof(id));
    dread(&len, sizeof(len));
    switch((unsigned int)id) {
      case CHUNK_RGB1:
           ReadRGBFloat(pc+len, rgb);
           break;
      case CHUNK_RGB2:
           ReadRGBByte (pc+len, rgb);
           break;
      default:
           dsetpos(pc+len);
    }
  }
}
```

● 读入纹理的名称。

纹理的名称也是一个 0 结尾串。

```
void ReadTextureNameChunk(dword p, H3dsTexture *texture) {
  int n;
  char name[16];
  n = 0;
  do {
    dread(&name[n++], 1);
  } while(name[n-1]!='\0' && n<sizeof(name));
  name[n-1]='\0';
  strcpy(texture->texturename, name);
}
```

● 读入纹理。

这里只读入纹理的名称，其他信息全部忽略，用户若有兴趣，可以自己将其补充

完整。

```
void ReadTextureChunk(dword p, H3dsTexture *texture) {
  unsigned short id;
  dword len, pc;
  while((pc=dgetpos()) < p) {
    dread(&id, sizeof(id));
    dread(&len, sizeof(len));
    switch((unsigned int)id) {
      case CHUNK_MAPFILENAME:
        ReadTextureNameChunk(pc+len, texture);
        break;
      default:
        dsetpos(pc+len);
    }
  }
}
```

- 置材质为双面材质。

  假定若一个材质为双面的，则所有材质都是双面的，因此可以用一个全局变量来标识材质为双面的。

```
void ReadDoublesidedChunk(int *material_doublesided) {
  *material_doublesided = 1;
  doublesided = 1;
}
```

- 读入当前材质块的信息。

  材质块中的信息包含几种类型，它可以是其名称或其环境光成分、漫反射光成分、镜面反射光成分或者是纹理或双面光照属性等。各种子类型信息都以子块的形式加以定义。

```
void ReadMaterialBlocks(dword p)
{
  unsigned short id;
  dword len, pc;
  H3dsMat *materialobj = EnlargeMatObjArray();
  while((pc=dgetpos()) < p) {
    dread(&id, sizeof(id));
    dread(&len, sizeof(len));
    switch((unsigned int)id) {
    // 材质名称块
    case CHUNK_MATNAME:
      ReadMatChunk(pc+len, materialobj);
```

```
          break;
        // 材质的环境光成分
        case CHUNK_AMBIENT:
          ReadShadingChunk(pc+len, &materialobj->ambient);
          break;
        // 材质的漫反射光成分
        case CHUNK_DIFFUSE:
          ReadShadingChunk(pc+len, &materialobj->diffuse);
          break;
        // 材质的镜面反射光成分
        case CHUNK_SPECULAR:
          ReadShadingChunk(pc+len, &materialobj.>specular);
          break;
        // 纹理子块
        case CHUNK_TEXTURE:
          ReadTextureChunk(pc+len, &materialobj.>texture);
          break;
        // 是否使用双面材质
        case CHUNK_DOUBLESIDED:
          ReadDoublesidedChunk(&materialobj.>doublesided);
          break;
        // 其他类型的子块，跳过去
        default:
          dsetpos(pc+len);
          break;
        }
      }
    }
```

- 读入形体对象块及材质。

```
void ReadObjMeshBlocks(dword p)
{
  unsigned short id;
  dword len, pc;

  while((pc=dgetpos()) < p) {
    dread(&id, sizeof(id));
    dread(&len, sizeof(len));
    switch((unsigned int)id) {
      case CHUNK_MATERIAL:
```

```
            ReadMaterialBlocks(pc+len);
            break;
          case CHUNK_OBJBLOCK:
            ReadObjBlocks(pc+len);
            break;
          default: dsetpos(pc+len);
        }
      }
    }
```

- 读入物体主块。

  3DS 中共定义了 2 种主块，一种是编辑器主块，另一种是关键帧主块，由于这里只是要将物体的形体读入并显示，而不涉及任何动画信息的处理，因此跳过关键帧信息块。若想要在程序中加入对关键帧的支持，也可以从这里开始。

```
void ReadMainBlocks(dword p)
{
  word id;
  dword len, pc;

  while((pc=dgetpos()) < p) {
    dread(&id, sizeof(id));
    dread(&len, sizeof(len));
    switch((unsigned int)id) {
      case CHUNK_OBJMESH:
        ReadObjMeshBlocks(pc+len);
        break;
      case CHUNK_KEYFRAMER:
        // 此处可加入对关键帧信息的读取函数
      default: dsetpos(pc+len);
    }
  }
}
```

- 从文件中读入一个 H3dsScene 数据结构。

  这里是 3DS 场景读取的主函数，函数中首先从文件中读入 3DS 文件标识，即基本块标识，若标识不正确，则说明不是一个合法的 3DS 文件。否则按 3DS 文件的组织，读入其主块信息。

```
H3dsScene *HRead3dsScene(void *ptr, int what, dword size)
{
  int retval = setjmp(EnvState);
```

```
    if(ptr==0)
      return 0;

    if(what==0)
      InFile=(FILE *) ptr;
    else
      return 0;
    //  初始化全局数据结构 Scene，并将其初值置为 0
    if  ((Scene=(H3dsScene *) malloc(sizeof(H3dsScene)))==0)
      return 0;
    memset(Scene,  0,  sizeof(H3dsScene));
    Scene->material = 0;
    Scene->NumMaterials = 0;
    if(retval==0) {
       //  返回地址集合，开始装入 3DS 数据
      word id;
      dword len,  pc;
      //  读入 3DS 文件标识
      pc=dgetpos();
      dread(&id,  sizeof(id));
  dread(&len,  sizeof(len));
      //  标识不正确，非 3DS 文件
      if((int)id!=CHUNK_MAIN){
      //  将分配到的空间释放
        HFree3dsScene(Scene);
              return 0;
      }
      ReadMainBlocks(pc+len);
    }
    else {
       //  读取数据出错，也将分配到的空间释放
      HFree3dsScene(Scene);
      return 0;
    }
      return Scene;
  }
```

- 将内部数据结构填充到一个易于读取的数据结构中。

```
void convert_to_outgl() {
  int x, y, z;
```

```
    int faceindex;
    int current;
    float32 swap;
    RGB ambient, diffuse, specular;
    VERTEX v;
    TRIANGLE tri;
    char *name;
    //  如果模型使用默认材质，则将默认材质定义为一个全白色的材质
    if (defaultmaterialwarning) {
      ambient.red = 1.0;
      ambient.green = 1.0;
      ambient.blue = 1.0;
      ambient.alpha = 1.0;
      diffuse.red = 1.0;
      diffuse.green = 1.0;
      diffuse.blue = 1.0;
      diffuse.alpha = 1.0;
      specular.red = 1.0;
      specular.green = 1.0;
      specular.blue = 1.0;
      specular.alpha = 1.0;
OutGLAddMaterial(ambient, diffuse, specular,
                "WG_DEFAULT", "", 1);
    }
    //  将每个材质的名称及其属性填入 OutGL 结构中
    for (x=0; x<Scene->NumMaterials; x++) {
      H3dsMat *mat = &Scene->material[x];
      ambient.red = mat->ambient.red;
      ambient.green = mat->ambient.green;
      ambient.blue = mat->ambient.blue;
      ambient.alpha = 1.0;
      diffuse.red = mat->diffuse.red;
      diffuse.green = mat->diffuse.green;
      diffuse.blue = mat->diffuse.blue;
      diffuse.alpha = 1.0;
      specular.red = mat->specular.red;
      specular.green = mat->specular.green;
      specular.blue = mat->specular.blue;
      specular.alpha = 1.0;
```

```
      OutGLAddMaterial(ambient, diffuse, specular,
                          mat->name, mat->texture.texturename,
                          mat->doublesided);
    }
    // 再将每个顶点和多边形的信息填入 OutGL 结构中，需注意顶点的填入顺序
    current = 0;
    for (x=0; x<Scene->meshobjs; x++) {
      H3dsMeshObj *meshobj = &Scene->meshobjlist[x];
      // 交换 y, z 坐标
      for (y=0; y<meshobj->NumVerts; y++) {
        swap = meshobj->vertlist[y].y;
        meshobj->vertlist[y].y = meshobj->vertlist[y].z;
        meshobj->vertlist[y].z = swap;
      }
      // 将顶点信息填入 OutGL 结构中去
      for (y=0; y<meshobj->NumVerts; y++) {
        v.x = meshobj->vertlist[y].x;
        v.y = meshobj->vertlist[y].y;
        v.z = meshobj->vertlist[y].z;
        // 填入形体的纹理坐标信息
        if (meshobj->NumMaps) {
            v.u = meshobj->maplist[y].u;
            v.v = meshobj->maplist[y].v;
        } else {
            v.u = 0;
            v.v = 0;
        }
        OutGLAddVertex(v);
      }
      for (y=0; y<meshobj->NumBindings; y++) {
        name = meshobj->bindings[y].name;
        for (z=0; z<meshobj->bindings[y].NumFaces; z++) {
          // 将多边形反向，即三角形(0, 1, 2) => (0, 2, 1)
          faceindex = meshobj->bindings[y].faces[z];
          tri.p0 = meshobj->facelist[faceindex].p0 + current;

          tri.p2 = meshobj->facelist[faceindex].p1 + current;
          OutGLAddPolygon(tri, name);
        }
```

```
    }
    current += meshobj->NumVerts;
  }
  // 将 OutGL 中的所有内容包装起来
  OutGLFinishModel();
}
// 读入 3DS 文件，并将它的内容赋给多边形和物体类
int parse3ds (char *filename)
{
  FILE *inf;
  H3dsScene *scene;
  long size;
  doublesided = defaultmaterialwarning = 0;
  // 以二进制方式打开文件
  if(!(inf = fopen(filename, "rb"))) {
    printf("Failed to open %s\n", filename);
    return 1;
  }
  if(fseek(inf, 0, SEEK_END)) {
    printf("Error seeking %s\n", filename);
    fclose(inf);
    return 1;
  }
  if((size=ftell(inf)) == .1L) {
    printf("Error sizing %s\n", filename);
    fclose(inf);
    return 1;
  }
  rewind(inf);
  // 读入文件并将其内容放入 scene 中去
  if((scene = HRead3dsScene(inf, 0, size)) == 0) {
    fclose(inf);
    return 1;
  }
  fclose(inf);
  // 将内部数据结构转换为 outgl 的结构
  convert_to_outgl();
  HFree3dsScene(scene);
```

```
//查找材质的下标
int FindMaterialIndex(char *name) {
  unsigned int n;
  for (n=0; n<NumPolyLists; n++) {
    if (strcmp(name, PolyList[n].name) == 0) {
      return n;
    }
  }
  return -1;
}
往顶点列表中加入一个顶点
void OutGLAddVertex(VERTEX v)
{
  void *mem;
  mem = realloc(Vertices, sizeof(VERTEX) * (NumVertex+1));
  Vertices = (VERTEX *)mem;
  Vertices[NumVertex] = v;
  center.x += v.x;
  center.y += v.y;
  center.z += v.z;
  /* 向正在创建的中文组中加入一个“槽”，使之能够记载哪些三角形拥有共同的某个
顶点，加入这项内容的目的是用来创建顶点的法向量。*/
  mem = realloc(CommonTriangleList,
  sizeof(TRIANGLENODE **) * (NumVertex+1));
  CommonTriangleList = (TRIANGLENODE **)mem;
  CommonTriangleList[NumVertex] = 0;
  NumVertex++;
}
// 加入一个多边形的材质
void OutGLAddPolygon(TRIANGLE tri, char *MaterialName) {
  void *mem;
  int MatNum;
  POLYGONLIST *list;
  SURFACE TempSurface;
  TempSurface.p0 = tri.p0;
  TempSurface.p1 = tri.p1;
  TempSurface.p2 = tri.p2;
  // 忽略退化多边形
  if (CalculateSurfaceNormal(&TempSurface) == DEGENERATE_SURFACE)
```

```
    return;
  //  查找材质在多边形列表中的下标
  MatNum = FindMaterialIndex(MaterialName);
  list = &PolyList[MatNum];
  //  将一个新三角形加到多边形列表中
  mem = realloc(list->TriangleIndexList,
                 sizeof(SURFACE) * (list->NumTriangles+1));
  list->TriangleIndexList = (SURFACE *)mem;
  list->TriangleIndexList[list->NumTriangles].p0 = tri.p0;
  list->TriangleIndexList[list->NumTriangles].p1 = tri.p1;
  list->TriangleIndexList[list->NumTriangles].p2 = tri.p2;
  CalculateSurfaceNormal(&list->TriangleIndexList[list-NumTriang-
les]);
  list->NumTriangles++;
  if (PolyList[MatNum].doublesided) {
    //  将一个新三角形加到多边形列表中
    mem = realloc(list->TriangleIndexList,
                   sizeof(SURFACE) * (list->NumTriangles+1));
    list->TriangleIndexList = (SURFACE *)mem;
    list->TriangleIndexList[list->NumTriangles].p0 = tri.p0;
    list->TriangleIndexList[list->NumTriangles].p1 = tri.p2;
    list->TriangleIndexList[list->NumTriangles].p2 = tri.p1;
    CalculateSurfaceNormal(&list->TriangleIndexList[list->Num
Triangles]);
    list->NumTriangles++;
  }
}
//  给模型加入材质
void OutGLAddMaterial(RGB ambient, RGB diffuse, RGB specular,
                      char *name, char
                      *texname, int doublesided)
{
  void *mem;
  mem = realloc(PolyList, sizeof(POLYGONLIST) * (NumPolyLists+1));
  PolyList = (POLYGONLIST *)mem;
  strcpy(PolyList[NumPolyLists].name, name);
  strcpy(PolyList[NumPolyLists].texturename, texname);
  PolyList[NumPolyLists].ambient = ambient;
  PolyList[NumPolyLists].diffuse = diffuse;
```

```
    PolyList[NumPolyLists].specular = specular;
    PolyList[NumPolyLists].doublesided = doublesided;
    PolyList[NumPolyLists].NumTriangles = 0;
    PolyList[NumPolyLists].TriangleIndexList = 0;
    NumPolyLists++;
  }
/*OutGLFinishModel 函数在读入整个模型之后对模型的数据进行一些整理工作，其内
容包括：相找顶点的法向量（而不是前面求出的面法向量），将模型居中（即定位于（0,0,0）
位置），顶点数据放入数据中，并且编译显示列表。注意，在求项点法向量时，所用的方
法是：先求出该顶点周围的所有面的法向量，再根据每个面对顶点法向量的贡献大小加权
平均得到。*/
void OutGLFinishModel()
{
  unsigned int OriginalNumVertex;
  unsigned int NumPolygons;
  unsigned int i;
  OriginalNumVertex = NumVertex;
  printf("Total number of vertices in model:  %d\n",  NumVertex);
  printf("\nGenerating normals...\n");
  LinkVerticesToPolygons();
  CalculateVertexNormals();
  MakeTriangleIndexList();
  NumPolygons = 0;
  for (i=0;  i<NumPolyLists;  i++)
    NumPolygons += PolyList[i].NumTriangles;
  DisplayListNumber = glGenLists(1);
  OutGLCompileDisplayList();
}
/*  将所有法向量反向，因为有些 3DStudio 模型以顺时针方向为正方向  */
/*  若不反向，那么用 OpenGL 绘制时，有些面看上去就像是损坏了  */
void OutGLFlipNormals() {
  unsigned int i;
  flip ^= 1;
  for (i=0;  i<NumVertex;  i++) {
    Vertices[i].nx *= .1.0;
    Vertices[i].ny *= .1.0;
    Vertices[i].nz *= .1.0;
  }
  OutGLCompileDisplayList();
```

```
}
// 初始化描述的形体 OutGL 结构
void OutGLInitialize(char *name, int threshold, int strips, int c)
{
  NumVertex = 0;
  NumPolyLists = 0;
  Vertices = 0;
  CommonTriangleList = 0;
  PolyList = 0;
  center.x = center.y = center.z = 0;
  wireframe = flip = PolygonWinding = 0;
  strip = strips;
  centerModel = c;
}
//  创建显示列表
void OutGLCompileDisplayList() {
  unsigned int i, j;
  unsigned int current;
  glInterleavedArrays(GL_T2F_N3F_V3F, 0, Vertices);
  glNewList(DisplayListNumber, GL_COMPILE);
  for (i=0; i<NumPolyLists; i++) {
glMaterialfv(GL_FRONT , GL_AMBIENT , (GLfloat*)&PolyList[i].
ambient);
glMaterialfv(GL_FRONT , GL_DIFFUSE , (GLfloat*)&PolyList[i].
diffuse);
glMaterialfv(GL_FRONT , GL_SPECULAR , (GLfloat*)&PolyList[i].
specular);
    //  绘制三角形
    glDrawElements(GL_TRIANGLES, PolyList[i].NumTriangles*3,
 GL_UNSIGNED_INT,
 &PolyList[i].CrunchedTriangleIndexList[0]);
    }
    glEndList();
}
// WM_Paint 消息处理函数
void CRenderView: : OnPaint()
{
    CPaintDC dc(this); // 绘制描述
    HWND hWnd = GetSafeHwnd();
```

```
    HDC hDC = : : GetDC(hWnd);
    wglMakeCurrent(hDC, m_hGLContext);
    glClearColor( 1.0f, 1.0f, 1.0f, 0.0f ); //将屏幕背景清成白色
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    ModelCoordinates();
    OutGLDisplay();
    WorldCoordinates();
    // 使用了双缓存
    SwapBuffers(hDC);
}
int CRenderView: : OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView: : OnCreate(lpCreateStruct) == -1)
        return -1;
    HWND hWnd = GetSafeHwnd();
HDC hDC = : : GetDC(hWnd);
    if(SetWindowPixelFormat(hDC)==FALSE)
        return 0;
    if(CreateViewGLContext(hDC)==FALSE)
        return 0;
    // 默认模式
    glPolygonMode(GL_FRONT, GL_LINE);
    glPolygonMode(GL_BACK, GL_LINE);
    glShadeModel(GL_FLAT);
    glEnable(GL_NORMALIZE);
    // 光源和材质的属性
    GLfloat ambientProperties[]  = {0.7f, 0.7f, 0.7f, 1.0f};
    GLfloat diffuseProperties[]  = {0.8f, 0.8f, 0.8f, 1.0f};
    GLfloat specularProperties[] = {0.0f, 0.8f, 0.2f, 1.0f};
    glClearDepth( 1.0 );
    glLightfv( GL_LIGHT0, GL_AMBIENT, ambientProperties);
    glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuseProperties);
    glLightfv( GL_LIGHT0, GL_SPECULAR, specularProperties);
    glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0f);
    // 默认情况下激活光照
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    return 0;
}
```

```
//与Visual C++ 6.0事件的接口
BOOL CRenderDoc: : OnOpenDocument(LPCTSTR lpszPathName)
{
    int flip, PolygonWinding;
if (!CDocument: : OnOpenDocument(lpszPathName))
    return FALSE;
    threshold = 0;
    strip = 0;
    flip = 0;
    PolygonWinding = 0;
    center = 0;
    init();
    OutGLInitialize((char *)lpszPathName, threshold,
                    strip, center);
    if (parse3ds((char *)lpszPathName)) {
        printf("Can't parse %s\n", lpszPathName);
    }
    if (flip)
        OutGLFlipNormals();
    if (PolygonWinding)
        OutGLReverseWinding();
    CameraReset();
    return TRUE;
}
void CRenderDoc: : init(void) {
GLfloat global_ambient[4]= {0.1f, 0.1f, 0.1f, 1.0f};
 GLfloat light0ambient[4] = {0.3f, 0.3f, 0.3f, 1.0f};
 GLfloat light0diffuse[4] = {0.6f, 0.6f, 0.6f, 1.0f};
 GLfloat light0specular[4] = {0.6f, 0.6f, 0.6f, 1.0f};
 GLfloat light0pos[4] = {0.0f, 0.0f, 0.0f, 1.0f};

 glEnable(GL_DEPTH_TEST);
 glDepthFunc(GL_LESS);
 if (!strip) {
   glEnable(GL_CULL_FACE);
   glCullFace(GL_BACK);
   glFrontFace(GL_CCW);
}
 glShadeModel(GL_SMOOTH);
```

```
    glPolygonMode(GL_FRONT, GL_FILL);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);
    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light0ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light0specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light0pos);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glMateriali(GL_FRONT, GL_SHININESS, 100);
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
}
```

4. 形体的各种几何运算

定义如下几个函数:

- 计算面的法向量

```
static int CalculateSurfaceNormal(SURFACE *);
```

- 计算 2 个面夹角的余弦

```
static GLfloat cos_angle(SURFACE *, SURFACE *);
```

- 计算一个三角形在某个顶点处的夹角

```
static GLfloat angle(SURFACE *, GLuint);
```

- 计算 2 个三维点之间的距离

```
static double dist(VERTEX *, VERTEX *);
```

- 建立每个顶点的相关三角形列表

```
static void LinkVerticesToPolygons();
```

- 计算每个顶点的法向量

```
static void CalculateVertexNormals();
```

- 建立三角形列表，每个三角形中记载其顶点在顶点列表中的下标

```
static void MakeTriangleIndexList();
```

首先给出模块中用到的宏定义和一些全局变量:

```
#define JOB_DONE            0       // 表明是否完成
#define DEGENERATE_SURFACE 1        // 非退化三角形
```

以下是该模块中用到的全局变量:

```
unsigned int NumVertex;
VERTEX *Vertices;
unsigned int NumPolyLists;
POLYGONLIST *PolyList;
VERTEX center;
int DisplayListNumber;
```

```
TRIANGLENODE **CommonTriangleList;
```

5. 计算三角形法向量函数

计算三角形法向量的方法很简单，顺序地从三角形中找出两条有向边，计算其叉乘，得到的结果就是三角形的法向量，一般情况下还要对结果进行单位化。

```
int CalculateSurfaceNormal(SURFACE *Triangle) {
 NORMAL Normal;
 GLfloat Ai, Aj, Ak;
 GLfloat Bi, Bj, Bk;
 GLfloat sum, magnitude;
 //  求出三角形的 2 条有向边向量 A, B, i, j, k 分别是其 3 个坐标分量
 Ai = Vertices[Triangle->p1].x - Vertices[Triangle->p0].x;
 Aj = Vertices[Triangle->p1].y - Vertices[Triangle->p0].y;
 Ak = Vertices[Triangle->p1].z - Vertices[Triangle->p0].z;
 Bi = Vertices[Triangle->p2].x - Vertices[Triangle->p1].x;
 Bj = Vertices[Triangle->p2].y - Vertices[Triangle->p1].y;
 Bk = Vertices[Triangle->p2].z - Vertices[Triangle->p1].z;
 //  用叉乘来计算多边形的法向量
 Normal.n0 = Aj*Bk - Ak*Bj;
 Normal.n1 = Ak*Bi - Ai*Bk;
 Normal.n2 = Ai*Bj - Aj*Bi;
 //  计算法向量模的平方
 sum = (Normal.n0 * Normal.n0) +
      (Normal.n1 * Normal.n1) +
      (Normal.n2 * Normal.n2);
 //  只要三角形 3 个顶点不全部在一点上，法向量的模就不等于 0
 if (sum == 0)  /*  否则为退化三角形  */
   return DEGENERATE_SURFACE;
 magnitude = 1/(GLfloat)sqrt(sum);
 //  向量单位化
 Normal.n0 *= magnitude;
 Normal.n1 *= magnitude;
 Normal.n2 *= magnitude;
 //  将结果记入三角形中
 Triangle->n0 = Normal.n0;
 Triangle->n1 = Normal.n1;
 Triangle->n2 = Normal.n2;
 return JOB_DONE;
}
```

6. 计算两个三角形的法向量夹角的余弦

求两个三角形的法向量夹角的余弦，因为这里的两个法向量均已单位化，故直接计算其点积即可。

```
GLfloat cos_angle(SURFACE *n1, SURFACE *n2)
{
  GLfloat dot;
  dot = n1->n0 * n2->n0 + n1->n1 * n2->n1 + n1->n2 * n2->n2;
  return dot;
}
```

7. 计算一个三角形在某个顶点处的夹角

根据毕达哥拉斯定理，有：$c^2=a^2+b^2+2ab\cos\theta$，这样，一个三角形在某顶点处的夹角：$\theta = \mathrm{acos}(\,(c^2 - a^2 - b^2\,)\,/\,2ab\,)$。

```
GLfloat angle(SURFACE *triangle, GLuint vertex) {
  double result;
  double a, b, c;
  //  先判断是哪一个顶点
  if (triangle->p0 == vertex) {
    //  计算三条边长
    a = dist(&Vertices[triangle->p0], &Vertices[triangle->p1]);
    b = dist(&Vertices[triangle->p0], &Vertices[triangle->p2]);
    c = dist(&Vertices[triangle->p1], &Vertices[triangle->p2]);
  } else if (triangle->p1 == vertex) {
    a = dist(&Vertices[triangle->p1], &Vertices[triangle->p0]);
    b = dist(&Vertices[triangle->p1], &Vertices[triangle->p2]);
    c = dist(&Vertices[triangle->p2], &Vertices[triangle->p0]);
  } else if (triangle->p2 == vertex) {
    a = dist(&Vertices[triangle->p2], &Vertices[triangle->p0]);
    b = dist(&Vertices[triangle->p2], &Vertices[triangle->p1]);
    c = dist(&Vertices[triangle->p0], &Vertices[triangle->p1]);
  }
  result = ( c * c - a * a - b * b )/ ( 2 * a * b );
  result = acos(result);
  return (GLfloat) result;
}
```

8. 计算两个三维点之间的距离

```
double dist(VERTEX *v1, VERTEX *v2) {
  double dx, dy, dz;
  double result;
  dx = fabs((v1->x - v2->x)*(v1->x - v2->x));
  dy = fabs((v1->y - v2->y)*(v1->y - v2->y));
  dz = fabs((v1->z - v2->z)*(v1->z - v2->z));
  result = sqrt(dx + dy + dz);
  return result;
}
```

9. 建立每个顶点的相关三角形列表

建立每个顶点的相关三角形列表，使用的方法是：将三角形加入其每一个顶点的三角形列表中，处理完所有三角形之后，就建立了所有顶点的三角形列表。

```
static void LinkVerticesToPolygons() {
  unsigned int i, j;
  SURFACE *Triangle;
  TRIANGLENODE *TempPtr;
  for (i=0; i<NumPolyLists; i++) {
    for (j=0; j<PolyList[i].NumTriangles; j++) {
      // 将每个三角形加入其三个顶点的相关三角形列表
      Triangle = &PolyList[i].TriangleIndexList[j];
      // 第一个点
      TempPtr = (TRIANGLENODE *) malloc(sizeof(TRIANGLENODE));
      TempPtr->triangle = Triangle;
      TempPtr->next = CommonTriangleList[Triangle->p0];
      CommonTriangleList[Triangle->p0] = TempPtr;
      // 第二个点
      TempPtr = (TRIANGLENODE *) malloc(sizeof(TRIANGLENODE));
      TempPtr->triangle = Triangle;
      TempPtr->next = CommonTriangleList[Triangle->p1];
      CommonTriangleList[Triangle->p1] = TempPtr;
      // 第三个点
      TempPtr = (TRIANGLENODE *) malloc(sizeof(TRIANGLENODE));
      TempPtr->triangle = Triangle;
      TempPtr->next = CommonTriangleList[Triangle->p2];
      CommonTriangleList[Triangle->p2] = TempPtr;
    }
```

```
    }
  }
```

10. 计算每个顶点的法向量

计算每个顶点的法向量有多种方法，可以简单地将周围三角形的法向量直接相加得到，也可以由其周围三角形加权平均得到。权值的取法也有多种，可以是每个三角形的面积，也可以是每个三角形在该点处的夹角。在本实例中就是用每个三角形在该点处的夹角作为权值的方法来求顶点的法向量。

```
static void CalculateVertexNormals() {
  void *mem;
  unsigned int i;
  int NewVertexAlreadyMade;
  int NewVertex;
  int NumOriginalVertex;
  GLfloat VertexAngle;
  GLfloat TotalAngles;
  TRIANGLENODE *TempPtr;
  TRIANGLENODE *ThisSurface;
  GLfloat sum, magnitude;

  NumOriginalVertex = NumVertex;
  NewVertex = 0;
  // 计算每个顶点的法向量
  for (i=0; i<NumVertex; i++) {
    Vertices[i].nx = Vertices[i].ny = Vertices[i].nz = 0;
    TotalAngles = 0;
    NewVertexAlreadyMade = 0;
    ThisSurface = CommonTriangleList[i];
    TotalAngles = 0;
//  遍历该顶点的所有相关三角形
    while (ThisSurface) {
        //  计算该三角形对顶点法向量的贡献
     VertexAngle = angle(ThisSurface.>triangle, i);
     Vertices[i].nx += ThisSurface->triangle->n0 * VertexAngle;
     Vertices[i].ny += ThisSurface->triangle->n1 * VertexAngle;
     Vertices[i].nz += ThisSurface->triangle->n2 * VertexAngle;
     TotalAngles += VertexAngle;
     ThisSurface = ThisSurface->next;
    }
```

```
    // 计算顶点法向量，即周围三角形法向量的加权平均
    Vertices[i].nx /= TotalAngles;
    Vertices[i].ny /= TotalAngles;
    Vertices[i].nz /= TotalAngles;
    // 再将其单位化
    sum = ((Vertices[i].nx * Vertices[i].nx) +
    (Vertices[i].ny * Vertices[i].ny) +
    (Vertices[i].nz * Vertices[i].nz));
    magnitude = 1/(GLfloat)sqrt(sum);
    Vertices[i].nx *= magnitude;
    Vertices[i].ny *= magnitude;
    Vertices[i].nz *= magnitude;
  }
  if (centerModel) {
    // 调整模型的顶点使其中心位于(0, 0, 0)
    center.x /= NumOriginalVertex;
    center.y /= NumOriginalVertex;
    center.z /= NumOriginalVertex;
    for (i=0; i<NumVertex; i++) {
      Vertices[i].x-= center.x;
      Vertices[i].y-= center.y;
      Vertices[i].z-= center.z;
    }
  }
}
```

11. 建立三角形列表

对于每个三角形记载其每个顶点在顶点列表中的下标。

```
static void MakeTriangleIndexList() {
  unsigned int i, j;
  for (i=0; i<NumPolyLists; i++) {
PolyList[i].CrunchedTriangleIndexList =(TRIANGLE *) malloc(sizeof
(TRIANGLE)* (PolyList[i].NumTriangles+1));
    for (j=0; j<PolyList[i].NumTriangles; j++) {
      PolyList[i].CrunchedTriangleIndexList[j].p0 =
          PolyList[i].TriangleIndexList[j].p0;
      PolyList[i].CrunchedTriangleIndexList[j].p1 =
           PolyList[i].TriangleIndexList[j].p1;
      PolyList[i].CrunchedTriangleIndexList[j].p2 =
```

```
                PolyList[i].TriangleIndexList[j].p2;
            }
        }
    }
```

12. **一些必要的矩阵运算**

在三维图形中对形体进行操作，免不了要进行一些矩阵运算，例如，世界坐标到局部坐标的互相转换，以及初始化相机位置等。首先定义 2 个数组，用于保存各种几何操作的当前状态。

```
GLfloat m[16];
GLfloat localm[16];
```

将它们分别看成一个按行优先顺序组织的矩阵，每 4 个数据为一行，这样便形成了一个 4 行 4 列的矩阵。分别将上述 2 个矩阵看成世界坐标系矩阵和局部坐标系矩阵，以下列出程序中用到的矩阵运算的几个函数：

- 世界坐标转换为局部坐标

```
void world_to_local()
{
  glGetFloatv(GL_MODELVIEW_MATRIX, m);
  glLoadIdentity();
}
```

- 局部坐标转换为世界坐标

```
void local_to_world(){
  glMultMatrixf(m);
}
```

- 设置为局部坐标系

```
void local()
{
  glPushMatrix();
  glLoadMatrixf(localm);
}
```

- 设置为世界坐标系

```
void world()
{
  glGetFloatv(GL_MODELVIEW_MATRIX, localm);
  glPopMatrix();
}
```

- 设置为模型坐标系

```
void ModelCoordinates()
{
```

```
  glPushMatrix();
  glMultMatrixf(localm);
}
```

- 设置为世界坐标系

```
void WorldCoordinates()
{
  glPopMatrix();
}
```

- 初始化相机位置

```
void CameraReset()
  glLoadIdentity();
  glGetFloatv(GL_MODELVIEW_MATRIX,  localm);
  glTranslatef(0,  0, -200);
}
```

第 10 章　位图、字体和图像编程

由于文字信息有强大的抽象能力和图形信息所无法替代的描述功能，因此在绘图程序中加入文字提示信息是完全必要的。有时用户还希望在图形绘制程序中使用一些背景图像，从而涉及到图像在绘图程序中的使用。本章中主要就是讨论以上内容在 OpenGL 中的实现。

10.1　位图和字体

OpenGL 中位图就是 1 个由 0 和 1 组成的矩形阵列，起到窗口中对应矩形位置的绘制掩模的作用。现假定绘制一幅位图，当前颜色为红色。在位图中为 1 的地方，对应的像素将被红色像素取代(或与红色像素组合)。如果在位图中有个 0，对应像素的内容将不受影响。通常使用位图是为了在屏幕上绘制字符。

10.1.1　OpenGL 对字符显示的支持

OpenGL 只提供了绘制字符串和操纵字体的最低级的支持。命令和在屏幕上定位并绘制单个位图。另外，通过显示列表机制，可以用一个字符编码序列去索引相应的表示这些字符的位图序列。用户还得自己写一个例程来提供对位图、字体、字符串操纵的所需的支持。

参见示例 10.1，它实现在屏幕上绘制 3 个字母 F。如图 10.1 所示是作为位图的 F 和它对应的位图数据。

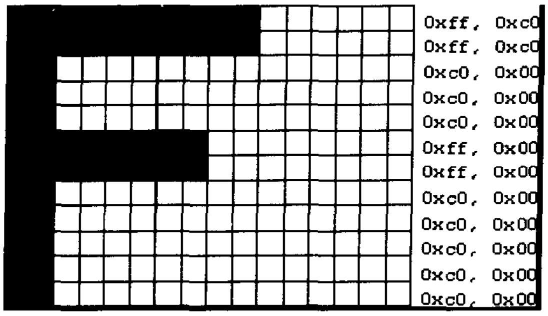

图10.1　字母F的位图数据

例 10.1　绘制一个位图表示的字符 drawf.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
```

```
GLubyte rasters[24] = {
    0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
    0xc0, 0x00, 0xff, 0x00, 0xff, 0x00, 0xc0, 0x00,
    0xc0, 0x00, 0xc0, 0x00, 0xff, 0xc0, 0xff, 0xc0};
void myinit(void)
{
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glClearColor(0.0, 0.0, 0.0, 0.0);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glRasterPos2i (20.5, 20.5);
    glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glFlush();
}
void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, w, 0, h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(0, 0, 500, 500);
    auxInitWindow(argv[0]);
    myinit();
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
}
```

在图 10.1 中，注意字符 F 的可见部分有大约 10 位(bit)宽。位图数据总是以 8 位的若干倍的块存储，但位图实际的宽度却不必是 8 的倍数。这些组成位图的位从左下角开始绘制：

首先，底线被绘制，然后绘制它上面的那条线，依次类推。可以从代码中了解到，位图以这个顺序存在内存中，光栅矩阵以 0xc0，0x00，0xc0，0x00 从 F 的底部两行开始，直到代表最上 2 行的 0xff，0xc0，0xff，0xc0。

例 10.1 中令人感兴趣的命令是 glRasterPos 和 glBitmap；将在 10.1.2 “当前光栅位置”一节中仔细讨论这两个函数的用法。

10.1.2 当前光栅位置

当前光栅位置是下一个位图(或图像)绘制的原点。在例 10.1 中，光栅位置设为(20，20)，那是所绘制 F 的左下角：

```
glRasterPos2i(20, 20);
void glRasterPos{234}{sifd}{v}(TYPE x, TYPE y, TYPE z, TYPE w);
```

设定当前光栅位置。参数 x，y，z 和 w 说明了光栅位置的坐标。

光栅位置的坐标变换为屏幕坐标的过程与命令所提供的坐标的变换过程完全一样(即，经模型视见和透视变换矩阵)。变换之后，它们或者定义 1 个屏幕上窗口中 1 个有效的位置，或者因为变换后位于视区之外而被裁剪掉。如变换后的点被裁剪掉，则当前光栅位置无效。

要获得当前光栅位置，可以用 GL_CURRENT_RASTER_POSITION 作为参数调用 gletFloatV 函数。函数的第二个参数必须是一个指向内存阵列的指针，这个阵列要足以容纳 4 个浮点值。以 GL_CURRENT_RASTER_POSITION_VALID 作为第一个参数调用可以，决定当前光栅位置是否有效。

10.1.3 绘制位图

一旦设定了光栅位置，就可以用命令去绘制那些数据了。

```
void glBitmap(GLsizei , width GLsizei height, GLfloat xbo, GLfloat
ybo, GLfloat xbi, GLfloat ybi, const GLubyte 8bitmap);
```

上述函数用于绘制位图，其中参数 bitmap 是指向位图图像的指针位图的原点被放在最接近当前光栅位置的地方。如果当前光栅位置无效，什么也不绘制。而光栅位置保持无效。参数 width 和 height 说明位图的宽度和高度，以像素为单位。宽度不必是 8 的倍数，尽管数据是保存在 8 位无符号字符中的(在 10 中，如果有 10 个位以外的多余的位是没有关系的，因为是以宽度 10 调用的，每行只有 10 位被绘制)。用 xbo 和 ybo 定义位图的原点(正值将原点移向右上，负值将它移向左下)，xbi 和 ybi 说明在位图光栅化后加给光栅位置的增量，如图 10.2 所示。

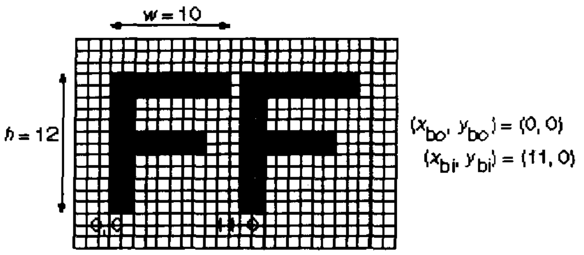

图10.2　一个位图及其相关参数

由于可以把位图的原点放在任意位置，使那些伸展到原点以下或原点左边的字符变得易于处理。

位图被绘制之后，当前光栅位置将会在 x 方向和 y 方向分别前进 bi 和 bi。对标准的拉丁(Latin)字体，bi 通常是 0.0，而 bi 是正值(因为连继的字符是从左到右绘制的)。对希伯莱字体(Hebrew)，字母是从右到左的，bi 通常将会是负值。连续的字符被垂直绘制成一列时，将令 bi 为 0，而 bi 为非负值。在图 10.2 中，每次 F 被绘制，当前光栅位置前进 12 个像素，在相邻字符间有 2 个像素的空间。

因为 bo，bo，bi，bi 是浮点值，字符的像素宽度不必是整数值。实际的字符将会在精确的像素边界上被绘制，但当前光栅位置仍保持浮点值，这样每个字符将会被绘制在尽可能接近它应该的位置。例如，在例 10.1 中的代码这样修改：bi 是 11.5 而不是 12，并且绘制更多的字符，字符间的空间将被改成 1~2 个像素之间，以取得对所要求的 1.5 个像素的最佳似近。

注意：位图不能用于可旋转字体，因为位图总是对齐帧缓存绘制的。

10.1.4　字体和显示列表

一些显示列表控制命令对绘制字符串有特殊的用处。这里所说的将用位图绘制的字符与用几何图元(点、线、多边形)绘制的等价。

一个字体通常由一个字符集，其中每个字符有一个特征数字(通常是 ASCII 码)，和一种绘制方法组成。对标准的 ASCII 字符集，大写字母 A 是数字 65，B 是 66，依此类推。在最简单的近似显示列表数字 65 将绘制一个 A，数字 66 绘制一个 B，依此类推。那么，要绘制字符串 68, 65, 66，只要执行相应的显示列表。可以用这种方式调用命令 glCallLists()。

```
void glCallLists(Glsizei n, Glenum type, const Glvoid *lists);
```

第一个参数，说明要绘制的字符个数，通常是 GL_BYTE，而不是一个字符代码数组。因为许多应用程序需要用不同的字体和尺寸绘制字符串，这种最简单的近似并不方便。相反，如果把 65 当作 A 而不管是哪个字体在起作用，可以强制在字体 1 中将 A、B、C 编码为 1065，1066，1067，而在字体 2 中为 2065，2066，2067，然而任何超过 256 的数字将不再适合 8 位字节(byte)。一个更好的解决方法是给字符串和选择显示列表的每个条目加一个偏移量。在这种情况下，字体 1 中 A，B，C 表示为 1065，1066，1067，而在字体 2 中可能是 2065，2066，2067。当用字体 1 绘制字符时，设定偏移量为 1000 并绘制显示列表 65，66，67。要用字体 2 绘制相同的字符串则将偏移量设为 2000 并绘制相同的显示列表。

在这种近似下，用命令设定偏移量。对前面的例子，它将以 1000 或 2000 作为唯一参数被调用。现在需要一串连续，未被使用的显示列表数字，可以调用 glGenLists()函数获得它。

```
GLuint glGenLists(Glsizei range);
```

这个函数返回 range 个显示列表标识符组成的块。返回的列表将全部标记为已被使用尽管它们是空的。这样，接下来对 glGenLists()的使用就绝不会返回相同的列表(除非在这之前已经明确地删除了它们)。如果用 4 作为参数并且如果返回 81，可以为字符使用显示列表标识符 81，82，83，84。如果找不到一个指定长度的未使用标识符块，它将返回 0 。

注意：命令使得删除与一个字体相关联的所有显示列表简化到只用一条指令。

许多美洲和欧洲字体只有一个很小的字体集(少于 256)，因此用不同的单字节代码代表每个字符是很容易的。其他字体中的亚洲字体可能要求一个巨大的字符集，每个字符一个字节编码是不可能的。通过的参数，OpenGL 允许字符串由 1，2，3，4 字节字符组成。这个参数可以取以下值中的一个：

```
GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT
GL_UNSIGNED_INT, GL_FLOAT, GL_2_BYTES, GL_3_BYTES, GL_4_BYTES
```

10.1.5 定义和使用一个完整的字体

本节中样本的程序用命令和显示列表机制定义了一个完整的光栅字体。一个完整的 ASCII 字体被定义为每个字符宽度相同。代码与例 10.2 很像，除了需要 95 个不同的位图(每个可打印一个 ASCII 字符，含空格)，并且每个都是显示列表中一条单独的命令。每个显示列表标识符与 ASCII 码相等，都被加上相同的偏移量。例 10.2 产生的输出如图 10.3 所示。

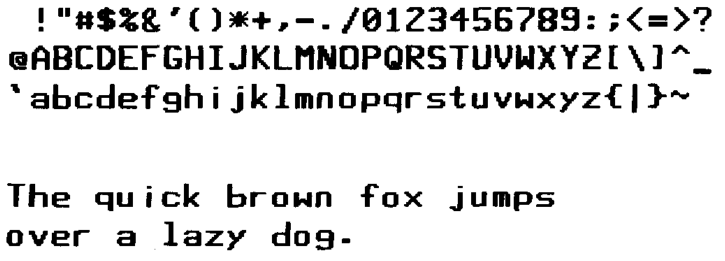

图10.3 定义一个完整的字体集

例 10.2 定义一个完整的字体集：font.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
GLubyte rasters[][13] = {
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00},
    {0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18,
```

```
0x18, 0x18, 0x18},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36,
0x36, 0x36, 0x36},
{0x00, 0x00, 0x00, 0x66, 0x66, 0xff, 0x66, 0x66, 0xff, 0x66,
0x66, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x7e, 0xff, 0x1b, 0x1f, 0x7e, 0xf8, 0xd8,
0xff, 0x7e, 0x18},
{0x00, 0x00, 0x0e, 0x1b, 0xdb, 0x6e, 0x30, 0x18, 0x0c, 0x76,
0xdb, 0xd8, 0x70},
{0x00, 0x00, 0x7f, 0xc6, 0xcf, 0xd8, 0x70, 0x70, 0xd8, 0xcc,
0xcc, 0x6c, 0x38},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18,
0x1c, 0x0c, 0x0e},
{0x00, 0x00, 0x0c, 0x18, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x30, 0x18, 0x0c},
{0x00, 0x00, 0x30, 0x18, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
0x0c, 0x18, 0x30},
{0x00, 0x00, 0x00, 0x00, 0x99, 0x5a, 0x3c, 0xff, 0x3c, 0x5a,
0x99, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0xff, 0xff, 0x18, 0x18,
0x18, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x60, 0x60, 0x30, 0x30, 0x18, 0x18, 0x0c, 0x0c, 0x06,
0x06, 0x03, 0x03},
{0x00, 0x00, 0x3c, 0x66, 0xc3, 0xe3, 0xf3, 0xdb, 0xcf, 0xc7,
0xc3, 0x66, 0x3c},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x78, 0x38, 0x18},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06,
0x03, 0xe7, 0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0x07, 0x03,
0x03, 0xe7, 0x7e},
{0x00, 0x00, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0xff, 0xcc, 0x6c,
0x3c, 0x1c, 0x0c},
```

```
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0xfe, 0xc0, 0xc0,
0xc0, 0xc0, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc0, 0xc0,
0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x18, 0x0c, 0x06, 0x03,
0x03, 0x03, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e, 0xe7, 0xc3,
0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x03, 0x7f, 0xe7, 0xc3,
0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x38, 0x38, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x1c, 0x1c,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x06, 0x0c, 0x18, 0x30, 0x60, 0xc0, 0x60, 0x30,
0x18, 0x0c, 0x06},
{0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0xff, 0xff, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x06, 0x0c,
0x18, 0x30, 0x60},
{0x00, 0x00, 0x18, 0x00, 0x00, 0x18, 0x18, 0x0c, 0x06, 0x03,
0xc3, 0xc3, 0x7e},
{0x00, 0x00, 0x3f, 0x60, 0xcf, 0xdb, 0xd3, 0xdd, 0xc3, 0x7e,
0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3,
0x66, 0x3c, 0x18},
{0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3,
0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0,
0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,
0xc7, 0xce, 0xfc},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0,
0xc0, 0xc0, 0xff},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0,
0xc0, 0xc0, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0,
0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3,
```

```
0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x18, 0x18, 0x7e},
{0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06,
0x06, 0x06, 0x06},
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8,
0xcc, 0xc6, 0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0,
0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff,
0xff, 0xe7, 0xc3},
{0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3,
0xf3, 0xe3, 0xe3},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3,
0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3, 0x66, 0x3c},
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3,
0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0,
0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x18, 0x18, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3,
0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3,
0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c,
0x66, 0x66, 0xc3},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c,
0x66, 0x66, 0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06,
0x03, 0x03, 0xff},
{0x00, 0x00, 0x3c, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x30, 0x30, 0x3c},
```

```
{0x00, 0x03, 0x03, 0x06, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0x30,
0x30, 0x60, 0x60},
{0x00, 0x00, 0x3c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
0x0c, 0x0c, 0x3c},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc3,
0x66, 0x3c, 0x18},
{0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18,
0x38, 0x30, 0x70},
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0x7f, 0x03, 0xc3, 0x7e, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0,
0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x7e, 0xc3, 0xc0, 0xc0, 0xc0, 0xc3, 0x7e, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x03, 0x03,
0x03, 0x03, 0x03},
{0x00, 0x00, 0x7f, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0x7e, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30,
0x30, 0x33, 0x1e},
{0x7e, 0xc3, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0x7e, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0,
0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x00,
0x00, 0x18, 0x00},
{0x38, 0x6c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x00,
0x00, 0x0c, 0x00},
{0x00, 0x00, 0xc6, 0xcc, 0xf8, 0xf0, 0xd8, 0xcc, 0xc6, 0xc0,
0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x18, 0x18, 0x78},
{0x00, 0x00, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xfe, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xfc, 0x00,
0x00, 0x00, 0x00},
{0x00, 0x00, 0x7c, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x7c, 0x00,
```

```
        0x00, 0x00, 0x00},
        {0xc0, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0x00,
        0x00, 0x00, 0x00},
        {0x03, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe0, 0xfe, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0xfe, 0x03, 0x03, 0x7e, 0xc0, 0xc0, 0x7f, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0x1c, 0x36, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30,
        0x30, 0x30, 0x00},
        {0x00, 0x00, 0x7e, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0xc3, 0xe7, 0xff, 0xdb, 0xc3, 0xc3, 0xc3, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0xc3, 0x66, 0x3c, 0x18, 0x3c, 0x66, 0xc3, 0x00,
        0x00, 0x00, 0x00},
        {0xc0, 0x60, 0x60, 0x30, 0x18, 0x3c, 0x66, 0x66, 0xc3, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0xff, 0x60, 0x30, 0x18, 0x0c, 0x06, 0xff, 0x00,
        0x00, 0x00, 0x00},
        {0x00, 0x00, 0x0f, 0x18, 0x18, 0x18, 0x38, 0xf0, 0x38, 0x18,
        0x18, 0x18, 0x0f},
        {0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
        0x18, 0x18, 0x18},
        {0x00, 0x00, 0xf0, 0x18, 0x18, 0x18, 0x1c, 0x0f, 0x1c, 0x18,
        0x18, 0x18, 0xf0},
        {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x8f, 0xf1, 0x60,
        0x00, 0x00, 0x00}
};
GLuint fontOffset;
void makeRasterFont(void)
{
    GLuint i;
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    fontOffset = glGenLists (128);
    for (i = 32; i < 127; i++) {
```

```
            glNewList(i+fontOffset, GL_COMPILE);
            glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, rasters[i.32]);
            glEndList();
        }
    }
    void myinit(void)
    {
        glShadeModel (GL_FLAT);
        makeRasterFont();
    }
    void printString(char *s)
    {
        glPushAttrib (GL_LIST_BIT);
        glListBase(fontOffset);
        glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
        glPopAttrib ();
    }
    void display(void)
    {
        GLfloat white[3] = { 1.0, 1.0, 1.0 };
        int i, j;
        char teststring[33];
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3fv(white);
        for (i = 32; i < 127; i += 32) {
        glRasterPos2i(20, 200 - 18*i/32);
        for (j = 0; j < 32; j++)
            teststring[j] = (char) (i+j);
            teststring[32] = 0;
            printString(teststring);
        }
        glRasterPos2i(20, 100);
        printString("The quick brown fox jumps");
        glRasterPos2i(20, 82);
        printString("over a lazy dog.");
        glFlush ();
    }
    void myReshape(GLsizei w, GLsizei h)
    {
```

```
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho (0.0, w, 0.0, h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

10.2 OpenGL 中生成图像的打印

对于 OpenGL 绘制的图像，若想将其打印出来，以前只能先截取屏幕，然后将剪贴板拷贝到某个绘图程序中才能进行各种常规的图像操作。这中方法固然可行，但经常这样做不免显得繁琐。因此，希望能够直接在程序中输出 OpenGL 绘制的图像，一方面为用户提供了更加方便的操作方式，另一方面直接在图形绘制程序中控制打印，可以有效地控制打印分辨率。

在介绍打印 OpenGL 生成的图像之前先讲解一下 OpenGL 中图像的绘制。在 10.1 “位图和字体”一节中重点介绍了字符的显示，但大多数情况下，用户更关心图像的显示。下面就先介绍这方面的内容。

10.2.1 OpenGL 中图像的绘制

1. 像素的读写

OpenGL 提供了基本像素读写函数：

```
void glReadPixels( GLint x, GLint y, GLsizei width, GLsizei height,
GLenum format, GLenum type, GLvoid *pixels );
```

函数参数(x，y)定义图像区域左下角坐标，width 和 height 分别描述图像的宽和高。

pixels 是一个指针，用于指向图像数据数组。format 指出数据元素格式(可以是索引或 R，G，B，A 值)。如表 10.1 所示为 format 的取值及其含义，如表 10.2 所示为 type 的取值及其数据类型。

表10.1 format的取值及其含义

取 值	含 义
GL_INDEX	单个颜色索引
GL_RGB	依次为红、绿、蓝分量
GL_RED	单个红色分量
GL_GREEN	单个绿色分量
GL_BLUE	单个蓝色分量
GL_ALPHA	单个 Alpha 值
GL_LUMINANCE_ALPHA	亮度和值
GL_STENCIL_INDEX	单个模板索引
GL_DEPTH_COMPONENT	单个深度分量

表10.2 type的取值及其数据类型

取 值	含 义
GL_UNSIGNED_BYTE	无符号 8 位整数
GL_BYTE	8 位整数
GL_BITMAP	无符号 8 位整数数组中单个数位
GL_UNSIGNED_SHORT	无符号 16 位整数
GL_SHORT	16 位整数
GL_UNSIGNED_INT	无符号 32 位整数
GL_INT	32 位整数
GL_FLOAT	单精度浮点数

类似地，写像素可以调用以下函数：

```
void glDrawPixels( GLsizei width, GLsizei height, GLenum format, GLenum
type, const GLvoid *pixels );
```

2. 像素拷贝

```
void glCopyPixels( GLint x, GLint y, GLsizei width, GLsizei height,
GLenum type );
```

这个函数很类似先用 glReadPixels()然后调用 glDrawPixels()，但是它只是拷贝，并不消耗系统内存，type 可以是：GL_COLOR、GL_STENCIL 或 GL_DEPTH。在拷贝前，type 要按如下方式转换成 format：

- 当 type 为 GL_DEPTH 或 GL_STENCIL 时，format 对应的应该是 GL_DEPTH_COMPONENT 或 GL_STENCIL_INDEX。
- 当 type 为 GL_COLOR 时，format 应该是 GL_RGB 或 GL_COLOR_INDEX。

3. 图像缩放

用下面一段函数来说明图像缩放的用法。

```
void glPixelZoom(GLfloat zoomx, GLfloat zoomy);
zoomx zoomy 是 X, Y 方向的缩放因子。缺省是 1.0
#include "glos.h"
#include <GL/gl.h>
#include <GL/glaux.h>
#include "windows.h"
void myinit(void);
void CALLBACK  display(void);
void CALLBACK  reshape(GLsizei w, GLsizei h);

void myinit(void)
{
   auxInitDisplayMode(AUX_SINGLE|AUX_RGBA);
   auxInitPosition(0, 0, 500, 500);
   auxInitWindow("sample1");
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
}

void CALLBACK reshape(GLsizei w, GLsizei h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
     gluOrtho2D(0.0, 15.0, 0.0, 15.0*(GLfloat)h/(GLfloat)w);
   else
     gluOrtho2D(0.0, 15.0*(GLfloat)w/(GLfloat)h, 0.0, 15.0);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void draw()
{
   //  绘制一个彩色三角形(2D)
```

```
    glBegin(GL_TRIANGLES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex2f(2.0, 3.0);
    glColor3f(0.0, 1.0, 0.0);
    glVertex2f(12.0, 3.0);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2f(7.0, 12.0);
    glEnd();
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

//  在屏幕上方绘制原始图像
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(4.0, 8.0, 0.0);
    glScalef(0.5, 0.5, 0.5);
    draw();
    glPopMatrix();

    int i;
    for(i=0; i<5; i++)
    {
        //  循环 5 次画出 5 个拷贝
        //  先定义显示缩放因子(递增)
        glPixelZoom(1.0+0.1*i, 1.0+0.1*i);
        // 再定义拷贝原点(越来越向右上)
        glRasterPos2i(1+i*2, i);
        //  图像拷贝
        glCopyPixels(160, 310, 180, 160, GL_COLOR);
    }
    glFlush();
}
void main(void)
{
    myinit();
```

```
        auxReshapeFunc(reshape);
        auxMainLoop(display);
    }
```

这个例子在屏幕上方中央绘制一个彩色三角形，然后在下面从左到右，依次绘制它的拷贝，拷贝位置不断向右上方向，而且(通过增加缩放因子)新的拷贝变 tj 越来越大。当然，后绘制的拷贝会遮盖前面的图形。介绍完 OpenGL 中图像的使用方法之后，在 10.2.2 中讲解对 OpenGL 中生成图像打印的基本步骤。

10.2.2　OpenGL 中生成图像打印的基本步骤

为了演示如何在程序中将 OpenGL 绘制的内容打印出来，设计了以下程序。首先将 OpenGL 绘制的内容输出到一个位图结构中去，再将绘制的内容通过.startDoc 函数等将位图的内容输出到打印机中。

能够用 OpenGL 产生绘制的窗口中都同时用到了两个结构：HGLRC 和 CDC，通常要打印的 OpenGL 画面的数据就存放在这两个结构中。可以将这两个结构封装到一起形成一个新结构，并用它们生成相关的视图。

注意： OnPrint()中调用了 OnDraw()。如果在 OnDraw()中再次调用 wglMakeCurrent()，就不会产生打印，而是将重新绘制视图。可以在 OnBeginPrinting ()和 OnEndPrinting () 中设置和恢复当前 OpenGL 设备。

为了确保所要绘制的内容能够被画到正确的窗口中，必须在 OnDraw()以外的某个地方调用 wglMakeCurrent，例如，在 OnActivateView ()和 OnUpdate ()里。

有时，在 OnActivateView ()里调用 wglMakeCurrent 就够了，但在有些情况下，例如，在 MDI 应用程序中当创建第 2 个 CMDIChildWnd 窗口时，无法使用相同的绘制描述表，因此需要考虑一些特殊的实现方法。

10.2.3　实例过程

1.　结构定义及函数原型

首先在 glbmp.h 中定义了程序中用到的 2 个结构 MATATTRIB 和 RENDER，前者定义了材质的各种属性，后者则定义了用于 OpenGL 绘制的存储结构，包括 2 个 HDC 句柄和 1 个 HGLRC 句柄，以及用于保存绘制图像的位图结构等。

```
typedef struct _MatAttrib
{
    GLfloat Ambient[4];
    GLfloat Diffuse[4];
    GLfloat Specular[4];
    GLfloat Emission[4];
    GLfloat Shineness[1];
```

```
} MATATTRIB;

typedef struct _Render
{
HDC hDC, hMemDC;
    HGLRC   hglRC;
HBITMAP hBm, hBmOld;
    RECT  bmRect;
    BYTE     biInfo[sizeof(BITMAPINFOHEADER)  +  256  *  sizeof
(RGBQUAD)];
    LPVOID lpBits;
}   RENDER;
void InitializeGlobal(HWND hWndDlg);
HBITMAP CreateDIBSurface(HWND hWndDlg);
BOOL PrepareDIBSurface(void);
void CreateRGBPalette(void);
void CleanUp(HWND hWndDlg);
void InitGLScene(HWND hWndDlg);
void DrawGLImage(HWND hWndDlg);
BOOL DoPrint(HWND hWnd);
BOOL CALLBACK AbortPrintJob(HWND hWndDlg, UINT message,
                            WPARAM wParam, LPARAM lParam);
BOOL CALLBACK DlgProc (HWND, UINT, WPARAM, LPARAM);
```

2. 包含文件、全局变量和宏定义

```
#include <windows.h>
#include <commctrl.h>
#include <tchar.h>
#include <string.h>
#include <stdlib.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include "resource.h"
#include "glbmp.h"

#define WIDTHBYTES(bits)  (((bits) + 31)/32 * 4)
HINSTANCE ghInst;   // 当前实例
MATATTRIB mat;
```

```
RENDER  render;
BOOL bAbort = FALSE;
HWND hDlgCancel;   // 取消打印对话框
```

3.　主程序：WinMain函数

在主程序中，首先用 ZeroMemory 函数将 render 和 mat2 个结构清零，然后调用 DialogBox 弹出主对话框。

```
int APIENTRY WinMain (HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPTSTR  lpszCmdLine,  int nCmdShow)
{
    ghInst = hInstance;
ZeroMemory(&render,  sizeof(RENDER));
ZeroMemory(&mat,  sizeof(MATATTRIB));
DialogBox(hInstance,  MAKEINTRESOURCE(IDD_MAINDLG),
     NULL,  (DLGPROC)DlgProc);
  return(0);
}
```

4.　对话框事件处理程序

对话框事件处理函数处理主对话框中的各种消息，这些消息包括初始化对话框消息和窗口重绘消息、菜单项消息等。在接收到初始化对话框消息后，程序进行一系列初始化工作，例如，装入图标、初始化全局数据和窗口定位等。在接收到窗口重绘消息后，用 BitBlt 函数在绘制区将 OpenGL 绘制的内容显示到窗口上。在接收到退出菜单项消息或关闭按钮消息时，将对话框关闭，并退出应用程序。若单击打印按钮，则调用打印过程实现图像打印。

```
BOOL CALLBACK DlgProc( HWND hDlg,  UINT message,
                       WPARAM uParam,  LPARAM lParam)
{
    TCHAR szVal[20];
    switch (message)
    {
case WM_INITDIALOG:   // 初始化对话框消息
SetClassLong(hDlg,  GCL_HICON,
          (LONG)LoadIcon(ghInst,  MAKEINTRESOURCE(IDI_ICON1)));
    InitCommonControls();
    InitializeGlobal(hDlg);
    {
        RECT rc;
```

```
GetWindowRect(hDlg, &rc);
MoveWindow(hDlg, 100, 100, rc.right.rc.left, rc.bottom.rc.top,
TRUE);
}
            return (TRUE);
  case WM_PAINT:
PAINTSTRUCT ps;
BeginPaint(hDlg, &ps);
BitBlt(render.hDC ,   render.bmRect.left ,   render.bmRect.top ,
render.bmRect.right, render.bmRect.bottom,
  render.hMemDC, 0, 0, SRCCOPY);
EndPaint(hDlg, &ps);
              break;
       case WM_COMMAND:
            if (LOWORD(uParam) == IDOK
            || LOWORD(uParam) == IDCANCEL) {
                  CleanUp(hDlg);
EndDialog(hDlg, TRUE);        // 退出对话框
                return (TRUE);
            }
              else
              {
                  switch (LOWORD(uParam))
                  {
case IDC_PRINT:
                          DoPrint(hDlg);
                          break;
                  }
              }
            break;
       }
}
```

5. 初始化全局数据

```
void InitializeGlobal(HWND hWndDlg)
{
    mat.Ambient[3] = mat.Diffuse[3] = 1.0f;
    mat.Specular[3] = mat.Emission[3] = 1.0f;
    mat.Diffuse[1] = mat.Specular[1] = 0.5f;
```

```
        mat.Emission[0] = 0.8f;
        mat.Emission[1] = 0.3f;
        mat.Shineness[0]= 15.0f;
    render.hDC = GetDC(GetDlgItem(hWndDlg, IDC_PICTURE));
        render.hMemDC = CreateCompatibleDC(render.hDC);
    GetClientRect(GetDlgItem(hWndDlg, IDC_PICTURE), &render.bmRect);
        render.hBm = CreateDIBSurface(hWndDlg);
    render.hBmOld = SelectObject(render.hMemDC, render.hBm);
        if (!PrepareDIBSurface())
    MessageBox(NULL, "The pixel format is not set properly", "Error",
    MB_OK);
        render.hglRC = wglCreateContext(render.hMemDC);
    wglMakeCurrent(render.hMemDC, render.hglRC);
        InitGLScene(hWndDlg);
        DrawGLImage(hWndDlg);
    }
```

6. 初始化OpenGL场景

```
void InitGLScene(HWND hWndDlg)
 {
   GLfloat light_ambient[] = { 0.1f, 0.1f, 0.1f, 1.0f };
   GLfloat light_diffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f };
   GLfloat light_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
   // 设置光源位置
   GLfloat light_position[] = { 1.0f, 1.0f, 1.0f, 0.0f };
   glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
   glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   // 激活光照
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   // 激活深度检测
   glDepthFunc(GL_LESS);
   glEnable(GL_DEPTH_TEST);
   glViewport(0, 0, render.bmRect.right, render.bmRect.bottom);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
```

```
    if (render.bmRect.right <= render.bmRect.bottom)
    glOrtho  (-2.5 , 2.5 , -2.5*(GLfloat)render.bmRect.bottom/基层
(GLfloat)render.bmRect.right,
        2.5*(GLfloat)render.bmRect.bottom/(GLfloat)render.bmRect.r
ight, -10.0, 10.0);
    else
    glOrtho    (-2.5*(GLfloat)render.bmRect.right/    (GLfloat)
render.bmRect.bottom, 2.5*(GLfloat)render.bmRect.right/(GLfloat)
render.bmRect.bottom, -2.5, 2.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}
```

7. 绘制OpenGL图像

清除窗口后，利用前面设置的材质属性设置绘制环境，然后在两个位置分别用辅助库绘制 1 个实心锥体和 1 个实心球体。在绘制过程中将鼠标设置为等待形状。

```
void DrawGLImage(HWND hWndDlg)
{
    HCURSOR hcurSave;
hcurSave = SetCursor(LoadCursor(NULL, IDC_WAIT));
glClearColor(0.0f, 0.0f, 1.0f, 0.0f);  //background clear color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMaterialfv(GL_FRONT, GL_AMBIENT, mat.Ambient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat.Diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat.Specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat.Shineness);
   glMaterialfv(GL_FRONT, GL_EMISSION, mat.Emission);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   glPushMatrix ();
   glRotatef (20.0f, 1.0f, 0.0f, 0.0f);
   glPushMatrix ();
   glTranslatef (-0.75f, -0.5f, 0.0f);
   glRotatef (270.0f, 1.0f, 0.0f, 0.0f);
   auxSolidCone (1.0f, 2.0f);
   glPopMatrix ();

   glPushMatrix ();
   glTranslatef (0.75f, 0.0f, -1.0f);
```

```
    auxSolidSphere (1.0f);
    glPopMatrix ();
    glPopMatrix ();
    glFlush();
     SetCursor(hcurSave);
}
```

8. **实现打印过程**

```
BOOL DoPrint(HWND hWnd)
{
     BOOL bRet = FALSE;
     HDC hPrnDC;
     DOCINFO di;
    PRINTDLG  pd;
memset(&pd,  0,  sizeof(PRINTDLG));
    pd.Flags = PD_RETURNDEFAULT|PD_RETURNDC;
    pd.hDevNames = NULL;
    pd.hDevMode = NULL;
    pd.lStructSize = sizeof(PRINTDLG);
    PrintDlg((LPPRINTDLG)&pd);
     hPrnDC = pd.hDC;
    if (!hPrnDC)
    {
        if (pd.hDevNames) GlobalFree(pd.hDevNames);
        if (pd.hDevMode) GlobalFree(pd.hDevMode);
        return bRet;
    }
    bAbort  = FALSE;
hDlgCancel   =   CreateDialog(ghInst ,    MAKEINTRESOURCE   (IDD_
PRINTCANCEL), hWnd,  (DLGPROC)AbortPrintJob);
ShowWindow(hDlgCancel,  SW_NORMAL);  //  这两条语句确保取消打印对话框
UpdateWindow(hDlgCancel);            //  在真正打印之前被显示
EnableWindow(hWnd,  FALSE);
    di.cbSize = sizeof(DOCINFO);
    di.lpszDocName=  "GlBmp";
    di.lpszOutput =  NULL;
if (StartDoc(hPrnDC,  &di) == SP_ERROR) goto Error;
    if (StartPage(hPrnDC) <= 0) goto Error;
StretchDIBits(hPrnDC,  0,  0,  GetDeviceCaps(hPrnDC,  HORZRES)/2,
```

```
                GetDeviceCaps(hPrnDC, VERTRES)/2,
            0, 0, render.bmRect.right, render.bmRect.bottom,
        render.lpBits, (BITMAPINFO *)render.biInfo, DIB_RGB_COLORS,
                SRCCOPY);
        if (EndPage(hPrnDC) <= 0) goto Error;
        if (EndDoc(hPrnDC) <= 0) goto Error;
        bRet = TRUE;
    Error:
    EnableWindow(hWnd, TRUE);
        DestroyWindow(hDlgCancel);
        DeleteDC(hPrnDC);
        if (pd.hDevNames) GlobalFree(pd.hDevNames);
        if (pd.hDevMode) GlobalFree(pd.hDevMode);
        return (bRet && !bAbort);
    }
```

9. 放弃打印任务

```
BOOL CALLBACK AbortPrintJob(HWND hWndDlg, UINT message,
     WPARAM uParam, LPARAM lParam)
{
    switch (message) {
case WM_COMMAND:
            bAbort = TRUE;
            return TRUE;
default:
            return FALSE;
    }
    UNREFERENCED_PARAMETER(lParam);
    UNREFERENCED_PARAMETER(uParam);
    UNREFERENCED_PARAMETER(message);
}
```

10. 清除各种动态内存

该函数删除 DIB 区、内存设备描述表和绘制 OpenGL 描述表。该函数在程序退出前调用，以确保程序退出后，申请的动态内存能得到彻底的释放。

```
void CleanUp(HWND hWndDlg)
{
    if (render.hglRC)
    {
```

```
wglMakeCurrent(NULL, NULL);
        wglDeleteContext(render.hglRC);
    }
    DeleteObject(render.hPal);
render.hBm = SelectObject(render.hMemDC, render.hBmOld);
    DeleteObject(render.hBm);
    DeleteDC(render.hMemDC);
ReleaseDC(GetDlgItem(hWndDlg, IDC_PICTURE), render.hDC);
}
```

11. 设置像素格式

设置像素格式是每个基于 Windows 的 OpenGL 程序所必备的。

```
BOOL PrepareDIBSurface(void)
{
   static PIXELFORMATDESCRIPTOR pfd = {
sizeof(PIXELFORMATDESCRIPTOR), // 结构的大小
1,                    // 版本号
    PFD_DRAW_TO_WINDOW|PFD_DRAW_TO_BITMAP|
   PFD_SUPPORT_OPENGL | PFD_SUPPORT_GDI,
PFD_TYPE_RGBA,   // RGBA 类型
24, // 24 位颜色深度
0, 0, 0, 0, 0, 0,
0, // 没有 alpha 缓存
0, 0, //  没有累积缓存
0, 0, 0, 0, 32, // 32 位 z.buffer
0, //  没有模板缓存
0, //  没有辅助缓存
PFD_MAIN_PLANE, // 主层
0,
0, 0, 0// 忽略层掩码
    };
   BOOL bRet = TRUE;
   int nIndex;
   pfd.cColorBits = GetDeviceCaps(render.hDC, PLANES) *
                 GetDeviceCaps(render.hDC, BITSPIXEL);
   if (!(nIndex = ChoosePixelFormat(render.hMemDC, &pfd)))
     bRet = FALSE; DescribePixelFormat(render.hMemDC, nIndex,
sizeof(PIXELFORMATDESCRIPTOR), &pfd);
   if (!SetPixelFormat(render.hMemDC, nIndex, &pfd))
```

```
    bRet = FALSE;
  return bRet;
}
```

10.2.4 程序执行结果

程序执行结果如图 10.4 所示，在场景中绘制了加了光照的球体和圆锥，Print 和 Close 按钮分别实现打印和关闭程序的功能。

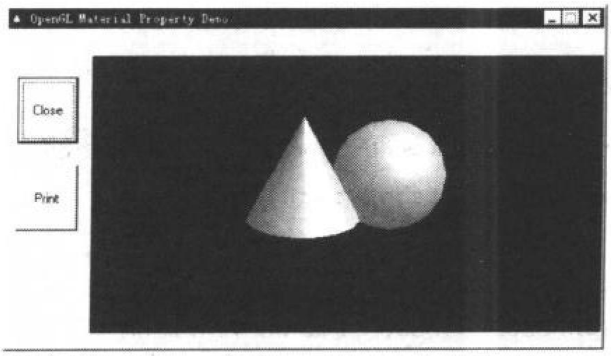

图10.4 打印OpenGL生成结果

第 11 章　纹理映射高级编程

在三维图形绘制程序中，纹理映射是广泛使用的。同时纹理映射也是一个相当复杂的过程，进行纹理映射需要做以下几步工作：

- 定义纹理。
- 控制滤波。
- 说明映射方式。
- 绘制场景给出顶点的纹理坐标和几何坐标。

注意： *纹理映射只能在 RGBA 模式下使用，不适用于颜色索引模式。*

在第 7 章中，在进行图形绘制时主要考虑的是使用各种几何图元的实颜色，或用图元中各个颜色的插值进行平滑明暗处理。而实际上纹理映射不仅可以节省绘制时间，而且也是很必要的。例如，对于一面砖墙，若不用纹理映射，则每块砖都必须用几个多边形来绘制，整面大墙需要用成千上万个多边形才能构造出来，还会由于过于平滑和规则而不够真实。事实上可以将这面墙看成一个大矩形，然后将一幅墙的照片贴上去，效果会更好。

纹理映射在其它方面也很有用，例如当绘制一大片地形时，地面上的草皮等由于过于复杂，用纹理映射的方法可以实现大体的效果。在地形漫游等大型应用程序中，纹理映射的方法经常被使用。

11.1　定义纹理

在最简单的情况下，纹理是单个图像。如同大多数图像一样，纹理通常是二维的，不过有时也使用一维纹理，在一些应用场合也使用三维纹理进行图形绘制。描述纹理的每个纹素可以由 1 个、2 个、3 个或 4 个元素构成，由 1 个（R，G，B，A）四元值的调制常数来表示任何内容。

在程序中可以调用以下函数定义二维纹理映射：

```
void glTexImage2D( GLenum target, GLint level, GLint components,
                   GLsizei width, GLsizei height, GLint border,
                   GLenum format, GLenum type, const GLvoid *pixels );
```

其中，target 指定目标纹理，只能是常数 GL_TEXTURE_2D；

level 表示多级分辨率的纹理图象的级数，若只有一种分辨率，level 为 0；

components 是从 1~4 的整数，1~4 分别表示，1：选择 R；2：选择 R A；3：选择 RGB；4：选择 RGBA；

width，height 是纹理的尺寸；分别对应纹理的宽度和高度，对某个数值 n，它必须是 $2^m+2*border$。

注意：纹理映射的最大宽度和高度取决于具体的 OpenGL 实现，最小为 64×64（或含边界时为 66×66）。假如设置 width 和 height 为 0，实际上相当于关闭了纹理映射。

border 指出边界的宽度，只能是 0 或 1，通常是 0。

format 和 type 描述映射格式和数据类型。它们与前面讲的 glDrawPixels()中的参数类似。

format 可以接受以下符号常量：

- GL_COLOR_INDEX
- GL_RGB
- GL_RGBA
- GL_RED
- GL_GREEN
- GL_BLUE
- GL_ALPHA
- GL_LUMINANCE
- GL_ LUMINANCE_ALPHA

type 可以接受以下符号常量：

- GL_TYPE
- GL_UNSIGNED_BYTE
- GL_SHORT
- GL_UNSIGNED_SHORT
- GL_INT
- GL_UNSIGNED_INT
- GL_FLOAT
- GL_BITMAP

level 参数用于指定不同分辨率的纹理，使用时自然要考虑到纹理滤波的问题。11.2 节就讨论有关控制纹理滤波的问题。

11.2 控制纹理滤波

纹理映射是正方形或矩形，但被映射到一个多边形或曲面上和变换为屏幕坐标后，纹理的各个纹素很少对应于最后屏幕图像的各个像素。在屏幕上的单个像素可对应于从很少部分的纹素到大量纹素的任何情况。在这些情况下，究竟应利用哪些纹素值和如何对这些值进行平均或内插是完全不清楚的。因此 OpenGL 提供了几种过滤选项，通过选择其中的一种来确定这些计算。这些选项给出速度和图象质量不同的折衷办法。另外，还可独立指定放大和缩小的滤波方法。

11.2.1 滤波

原始纹理图像是个方形图像，把它映射到奇形怪状的物体上，一般不可能图像上的一

个像素对应屏幕的一个像素。因此局部放大或缩小时，就要定义合适的滤波方式（以二维情况为例）：

```
void glTexParameter(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
void glTexParameter(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
```

前者是放大滤波（GL_TEXTURE_MAG_FILTER），后者是缩小滤波（GL_TEXTURE_MIN_FILTER）；另外，GL_NEAREST 是利用坐标最靠近像素中心的纹理元素，这有可能使图样走形，但计算速度快；GL_LINEAR 利用线形插值，效果好但计算量大。

11.2.2　重复与缩限

纹理映射可以重复映射或者缩限映射，重复映射时纹理可以在自己的坐标 S，T 方向重复。对于重复映射：

```
void glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
void glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
```

参数 GL_REPEAT 改为 GL_CLAMP，则缩限，所有大于 1 的纹理元素值置为 1。所有小于 0 的纹理元素值置为 0。0 的纹理元素值置为 0。

11.2.3　映射方式

处理纹理本身图案颜色和物体本身颜色的关系：

```
void glTexEnv{if}[v](GLenum target,GLenum pname,TYPE param);
```

target 必须是 GL_TEXTURE_ENV。

pname 是 GL_TEXTURE_ENV_MODE，则 param 可以是 GL_DECAL，GL_MODULATE 或 GL_BLEND，说明纹理值与原来颜色不同的处理方式。

pname 是 GL_TEXTURE_ENV_COLOR，则参数 param 是包含 4 个浮点数（R，G，B，A）的数组。这些值只在采用 GL_BLEND 纹理函数时才采用。

11.2.4　纹理坐标定义

坐标的定义：纹理图象是方形的，纹理坐标可定义成 s,t,r,q 坐标，仿照齐次坐标系的 x,y,z,w 坐标。

```
void glTexCoord{1234}{sifd}[v](TYPE coords);
```

设置当前纹理坐标，此后调用 glVertex*()所产生的顶点都赋予当前的纹理坐标。

11.2.5　自动产生纹理坐标

有时不需要为每个物体顶点赋予纹理坐标，可以使用：

```
void glTexGen{if}(GLenum coord,GLenum pname,TYPE param);
```

coord 为 GL_S，GL_T，GL_R 或 GL_Q，指明哪个坐标自动产生。pname 为 GL_TEXTURE_GEN_MODE 时，param 为常数 GL_OBJECT_LINEAR GL_EYE_LINEAR 或 GL_SPHERE_MAP，它们决定用哪个函数来产生纹理坐标；pname 为 GL_OBJECT_PLANE 或 GL_EYE_PLANE，param 是一个指向参数数组的指针。

例 11.1 举一个简单的例子

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glaux.h>
#include "windows.h"
void myinit(void);
void CALLBACK display(void);
void CALLBACK reshape(GLsizei w,GLsizei h);

// 创建纹理图像的子程序
#define TEXTUREWIDTH 64
#define TEXTUREHEIGHT 64
GLubyte Texture[TEXTUREWIDTH][TEXTUREHEIGHT][3];
void makeTexture(void)
void makeTexture(void)
{
    int i,j,r,g,b;
    for(i=0;i<TEXTUREWIDTH;i++)
    {
        for(j=0;j<TEXTUREHEIGHT;j++)
        {
            r=(i*j)%255;
            g=(4*i)%255;
            b=(4*j)%255;
            Texture[i][j][0 =(GLubyte)r;
            Texture[i][j][1 =(GLubyte)g;
            Texture[i][j][2 =(GLubyte)b;
        }
    }
}

void myinit(void)
{
    auxInitDisplayMode(AUX_SINGLE|AUX_RGBA);
    auxInitPosition(0,0,500,500);
```

```
    auxInitWindow("sample1");
    auxInitWindow("sample1");
    glClearColor(0.0,0.0,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);

    // 创建纹理图像的原始数据保存在 Texture[][][]中
    makeTexture();
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
    // 定义二维纹理
    glTexImage2D(GL_TEXTURE_2D,0,3,TEXTUREWIDTH,
              TEXTUREHEIGHT,0,GL_RGB,GL_UNSIGNED_BYTE,
              &Texture[0][0][0]);
    // 控制滤波
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR
          );
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR
          );

    // 说明映射方式
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);

    // 启用纹理模式
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_TEXTURE_2D);
    // glShadeModel(GL_FLAT);
}

void CALLBACK reshape(GLsizei w,GLsizei h)
{
    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // 定义立体视景体
    gluPerspective(60.0,1.0*(GLfloat)w/(GLfloat)h,1.0,30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0,0.0,-3.6);
```

```
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    glBegin(GL_QUADS);//绘制四边形
    // 先绘制正方形，用来显示实际未变形的纹理图样
    // 先绘制正方形，用来显示实际未变形的纹理图样
    glTexCoord2f(0.0,0.0);glVertex3f(-2.0,-1.0,0.0);
    glTexCoord2f(0.0,1.0);glVertex3f(-2.0,1.0,0.0);
    glTexCoord2f(1.0,1.0);glVertex3f(0.0,1.0,0.0);
    glTexCoord2f(1.0,0.0);glVertex3f(0.0,-1.0,0.0);

    // 绘制一个不规则四边形，用来显示纹理是如何随物体形状而变形的
    glTexCoord2f(0.0,0.0);
    glVertex3f(0.0,-1.0,0.0);
    glTexCoord2f(0.0,1.0);
    glVertex3f(0.0,1.0,0.0);
    glTexCoord2f(1.0,1.0);
    glVertex3f(1.41421,1.0,-1.41421);
    glTexCoord2f(1.0,0.0);
    glVertex3f(1.41421,-1.0,-1.41421);
    glEnd();

    glFlush();
}
void main(void)
{
    myinit();

    auxReshapeFunc(reshape);
    auxMainLoop(display);
}
```

从例 11.1 来看，除了纹理的定义和控制比较麻烦和不容易理解外，其应用是十分方便的。只须从纹理的坐标系选出合适点附在实际物体顶点上即可。对于复杂的纹理定义和控制，也可以自行改变一些参数，看看效果如何。例如，把 GL_LINEAR 改成 GL_NEAREST，则纹理明显变得粗糙，但计算结果却快多。

您也可以改动 glTexCoord2f()的参数，看看如何选定纹理的一部分（例子中是选定全

部纹理）来贴图。例如 1.0 改成 0.5，则选择实际纹理的左上 1/4 部分来贴图。在平面纹理贴图中，先定义了一个数组（一维或二维等）来定义纹理的数据，所以纹理本身是一个 N 维空间，有自己的坐标和顶点。在例 11.1 中，讲述了如何把纹理数据中的坐标和屏幕物体坐标相结合，就像把一块布料扯成合适的形状贴在物体表面一样。而例 11.1 中唯一没有使用的函数是纹理坐标的自动产生（最后一个给出的函数），它的作用是产生一个环境纹理，环境内的所有物体都赋予此纹理，很像一个特殊光源。

例 11.2　给出 1 个更复杂的纹理应用的例子来说明

```
#include "glos.h"
#include <GL/gl.h>
#include <GL/glaux.h>
#include "windows.h"
void myinit(void);
void CALLBACK display(void);
void CALLBACK reshape(GLsizei w,GLsizei h);

// 定义一个一维纹理的数据，从生成来看，保持红色、蓝色分量 255（MAX），
// 所以是渐变的紫色纹理，饱和度不断变化。
#define TEXTUREWIDTH 64
GLubyte Texture[3*TEXTUREWIDTH];
void makeTexture(void)
{
    int i;
    for(i=0;i<TEXTUREWIDTH;i++)
    {
        Texture[3*i =255;
        Texture[3*i+1 =255-2*i;
        Texture[3*i+2 =255;
    }
}
GLfloat sgenparams[]={1.0,1.0,1.0,0.0};

void myinit(void)
{
    auxInitDisplayMode(AUX_SINGLE|AUX_RGBA);
    auxInitPosition(0,0,500,500);
    auxInitWindow("sample1");
    glClearColor(0.0,0.0,0.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    // 创建纹理
```

```
        makeTexture();
        glPixelStorei(GL_UNPACK_ALIGNMENT,1);
        // 控制纹理
        glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
        glTexParameterf(GL_TEXTURE_1D,GL_TEXTURE_WRAP_S,GL_REPEAT);
        glTexParameterf(GL_TEXTURE_1D,GL_TEXTURE_MAG_FILTER,GL_LINEAR
                );
        glTexParameterf(GL_TEXTURE_1D,GL_TEXTURE_MIN_FILTER,GL_LINEAR
                );
        glTexImage1D(GL_TEXTURE_1D,0,3,TEXTUREWIDTH,0,
                  GL_RGB,GL_UNSIGNED_BYTE,Texture);
        // 唯一与前面例子不同的地方：启用纹理坐标自动产生，生成环境纹理
        // 纹理的方向 S
        glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
        glTexGenfv(GL_S,GL_OBJECT_PLANE,sgenparams);
        // 启用纹理
        glEnable(GL_TEXTURE_1D);
        glEnable(GL_TEXTURE_GEN_S);

        // 启用消隐
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LESS);
        glDepthFunc(GL_LESS);

        // 一些绘图控制，详细可参阅 VC5 联机帮助
        glEnable(GL_CULL_FACE);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_AUTO_NORMAL);
        glEnable(GL_NORMALIZE);
        glFrontFace(GL_CW);
        glCullFace(GL_BACK);
        glMaterialf(GL_FRONT,GL_SHININESS,64.0);
        // glShadeModel(GL_FLAT);
}

void CALLBACK reshape(GLsizei w,GLsizei h)
{
```

```
    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if(w<=h)
        glOrtho(-4.0,4.0,-4.0*(GLfloat)h/(GLfloat)w,
           4.0*(GLfloat)h/(GLfloat)w,-4.0,4.0);
    else
        glOrtho(-4.0*(GLfloat)h/(GLfloat)w,
            4.0*(GLfloat)h/(GLfloat)w,-4.0,4.0,-4.0,4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(30.0,1.0,0.0,0.0);
    //  利用辅助库函数绘制一个茶壶
    auxSolidTeapot(1.5);
    glPopMatrix();
    glFlush();
}

void main(void)
{
    myinit();
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(30.0,1.0,0.0,0.0);
    //  利用辅助库函数绘制一个茶壶
    auxSolidTeapot(1.5);
    glPopMatrix();
    glFlush();
}

void main(void)
{
```

```
        myinit();
        auxReshapeFunc(reshape);
        auxMainLoop(display);
    }
```

11.3 用位图文件创建纹理

纹理映射中所用到的数据既可以是程序生成的一组数据，也可以从外部文件中读取。对于从外部文件读取的方式，由于在微机上位图文件是最常见的一种图像格式，而且位图文件格式较为简单，读取起来也比较方便，因此可以选择这种文件作为纹理文件存储格式。当然，为了保持图形工作站的移植性，有时也需要用 rgb 格式的文件来创建纹理。

11.3.1 调用函数来创建纹理

使用纹理最简单的方法是利用 Windows 下提供的一个函数，来创建由这 2 种不同格式的图像文件产生的纹理。这个函数是：

```
AUX_RGBImageRec auxDIBImageLoad(LPCTSTR filename);
```

filename 参数是存储纹理的文件名，函数可以读取 2 种图像格式（即 bmp 格式和 rgb 格式）文件，并能够根据扩展名的不同，自动选择读取方法。AUX_RGBImageRec 是一个定义纹理数据的结构，其中最主要的 3 个域是 SizeX，SizeY 和 data，其中 SizeX 和 SizeY 分别存储纹理的宽和高，一般将其设置成相同的值。而 data 种则存储着具体的纹理数据。

如例 11.3 所示，是从一个程序中摘录的 1 段利用 AUX_RGBImageRec 结构来创建纹理的程序。它 pDoc->m_image 是一个 AUX_RGBImageRec 类型的结构，其中保存了文件中的纹理内容。

例 11.3 程序示例

```
int CTexView::CreateTexture(void)
{
            double xPow2, yPow2;
    int ixPow2, iyPow2;
    int xSize2, ySize2;
    static BYTE *pData;
    GLint glMaxTexDim;

            CTexDoc* pDoc = GetDocument();
            image=pDoc->m_image;
            glGetIntegerv(GL_MAX_TEXTURE_SIZE, &glMaxTexDim);

    if (image->sizeX <= glMaxTexDim)
```

```
        xPow2 = log((double)image->sizeX) / log((double)2.0);
    else
        xPow2 = log((double)glMaxTexDim) / log((double)2.0);

    if (image->sizeY <= glMaxTexDim)
        yPow2 = log((double)image->sizeY) / log((double)2.0);
    else
        yPow2 = log((double)glMaxTexDim) / log((double)2.0);

    ixPow2 = (int)xPow2;
    iyPow2 = (int)yPow2;

    if (xPow2 != (double)ixPow2)
        ixPow2++;
    if (yPow2 != (double)iyPow2)
        iyPow2++;

    xSize2 = 1 << ixPow2;
    ySize2 = 1 << iyPow2;

            if(pData!=NULL)
            free(pData);
    pData = (BYTE *)malloc(xSize2 * ySize2 * 3 * sizeof(BYTE));
    if (!pData)
        return 0;
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);

    gluScaleImage(GL_RGB, image->sizeX, image->sizeY,
                GL_UNSIGNED_BYTE, image->data,
                xSize2, ySize2, GL_UNSIGNED_BYTE,
                pData);

    if( tQuality == TEX_HIGH ) {
        gluBuild2DMipmaps(GL_TEXTURE_2D, 3, image->sizeX, image-
            >sizeY,
                GL_RGB, GL_UNSIGNED_BYTE, image->data);
    }
    else {
        glTexImage2D(GL_TEXTURE_2D, 0, 3, xSize2, ySize2,
```

```
                        0, GL_RGB, GL_UNSIGNED_BYTE, pData);
                }
            glEnable(GL_TEXTURE_2D);
            return(1);
    }
```

11.3.2 从文件中直接读写

有时为了更加方便地使用纹理文件的内容，会在程序中打开位图文件读取其中的纹理信息，并维护相应的纹理数据。这时，需要对 bmp 格式有 1 个大体了解。关于这方面的知识，只要知道对于一个 24 位颜色位图，从第 54 个字节开始就是位图数据(即存放着 rgb 颜色数组)即可。以下程序段就是从一个固定大小的文件中读取纹理内容的。

例 11.4 程序示例

```
void CRenderView::makeTexture(void) {
  int i, j, c;
  FILE *fp;
  unsigned char *image;
  int rc;

  fp = fopen("res\\tex002.bmp","rb");
  if(!fp) return;
  fseek(fp,54,SEEK_SET);
  image = (unsigned char *)malloc(32*32*3);
  rc=fread(image,sizeof(unsigned char),32*32*3,fp);
  fclose(fp);
  for(i=0; i<WIDTH; i++) {
     for(j=0; j<WIDTH; j++) {
      c = rand()%16+128;
      texImg[i][j][0] = (GLubyte) *(image+i*WIDTH*3+j*3);
      texImg[i][j][1] = (GLubyte) *(image+i*WIDTH*3+j*3+1);
      texImg[i][j][2] = (GLubyte) *(image+i*WIDTH*3+j*3+2);
      texImg[i][j][3] = (GLubyte) 255;
     }
  }
}
```

注意： 由于没有读取位图文件头中定义图像的大小等信息，因此需要事先确定位图的大小以及确保图像是 24 位。若有一点不符，则程序一定会出错。

第 12 章　各种特殊效果的实现

本章讲解以下内容：

- 如何对颜色进行融合，以使物体产生透明效果。
- 如何通过反走样，对线和多边形的锯齿状进行平滑处理。
- 如何创建逼真的大气雾化效果的场景。

为了使场景具有真实感，仅有前面所介绍的对几何模型的定义、变换、着色和明暗处理还远远不够，还需要用到一些能增添精细效果的技术。

- 颜色融合(blending)方法：通过指定如何将源和目的的颜色值相结合的融合函数，使最后产生的效果为部分场景成为半透明。
- 反走样(antialiasing)方法：通过改变点、线、多边形周围像素的颜色，使其平滑化，达到消除锯齿效果。
- 雾化(fog)：根据离视点的距离，计算对象的颜色值来建立深度的视觉。其效果与现实生活一样，当物体较远时背景较淡。

12.1　融　　合

前面提到的颜色的 RGBA 模式，其中的 A(即α 值)，当时总设置为 1.0。下面介绍的颜色融合技术则是通过设置不同的α 值来表现物体的透明度。当利用 glColor*()和利用 glClearColor()来指定背景色时，以及指定材质或光源亮度的光照参数时，可以指定α值。RGB 值决定显示器上像素的颜色，而 A 值则表示不透明度。融合操作可看作是用片段值中的 RGB 值分量表示颜色，A 分量表示不透明度。当融合被激活时，α 值用于融合正在处理的片段中的颜色值和已在帧缓存中的像素。融合发生在已光栅化场景和转换片段值之后，将最后像素写入帧缓存之前。α 值也能用于α 检验，根据α 值来确定接受或放弃一个片段值。

不进行融合时，每个新的片段值覆盖缓存中已有的颜色值，其效果相当于片段值是不透明的。进行融合时，可以控制已有的颜色值与新的片段值如何结合，利用α 融合建立一个半透明的片段值，让某些已有的颜色部分呈现，从而达到透明效果。

α 值表示不透明度，也就是说，如果物体表面透明或半透明，就要求不透明度比不透明表面低，即设置较小的α 值。例如，通过玻璃杯看杯中的铅笔，所看到的颜色，部分来自玻璃杯，部分来自玻璃杯中的铅笔。根据玻璃杯的透射特性两者有不同的比例。生活中，半透明现象是很常见的。

12.1.1　源和目的因子

在进行颜色融合过程中，新的颜色值与存储在缓冲区中的相应像素的颜色值相结合需要经过两个步骤。首先，指定如何计算源和目的因子，这些因子表示为 RGBA4 元组，它们依次乘以源和目的中的 R、G、B 和 A 分量。然后，再将两组 RGBA 四元组的对应分量相加。用数学形式表示，设源和目的融合因子依次为(S_r，S_g，S_b，S_a)和(D_r，D_g，D_b，D_a)，用 S 和 D 加下标表示源和目的的 RGBA 值。最后，经过颜色融合后的 RGBA 值为：

$(R_sS_r+R_dD_r,\ G_sS_g+G_dD_g,\ B_sS_b+B_dD_b,\ A_sS_a+A_dD_a)$

这个 4 元组的每个分量最后限制在[0，1]范围内。

下面讨论如何建立源和目的融合因子。利用 glBlendFunc()给出 2 个常数，一个指定如何计算源因子，另一个指定如何计算目的因子。要使融合起作用，必须使颜色融合激活，即调用：

```
glEnable(GL_BLEND);
```

当要使颜色融合不起作用时，调用：

```
glDisable(GL_BLEND);
```

当调用 glBlendFunc 函数时，若将源参数指定为 GL_ONE，同时目的参数指定为 GL_ZERO，则与调用 glDisable(GL_BLEND)结果一样。

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
```

函数的功能是控制正在处理的片段值与帧缓存中的已有颜色值如何结合，sfactor 参数表示源颜色值所占比例，dfactor 表示目的颜色值所占比例。这两个参数的可能取值及其含义如表 12.1 所示。

表12.1　源和目的因子及其含义

取　值	相关因子	计算后融合因子
GL_ZERO	源、目的	(0，0，0，0)
GL_ONE	源、目的	(1，1，1，1)
GL_DST_COLOR	源	(Rd，Gd，Bd，Ad)
GL_SRC_COLOR	目的	(Rs，Gs，Bs，As)
GL_ONE_MINUS_DST_COLOR	源	(1，1，1，1)-(Rd，Gd，Bd，Ad)
GL_ONE_MINUS_SRC_COLOR	目的	(1，1，1，1)-(Rs，Gs，Bs，As)
GL_SRC_ALPHA	源、目的	(As，As，As，As)
GL_ONE_MINUS_SRC_ALPHA	源、目的	(1，1，1，1)-(As，As，As，As)
GL_DST_ALPHA	源、目的	(Ad，Ad，Ad，Ad)
GL_ONE_MINUS_DST_ALPHA	源、目的	(1，1，1，1)-(Ad，Ad，Ad，Ad)
GL_SRC_ALPHA_SATURATE	源	(f，f，f，1)-min(As，1-Ad)

12.1.2 颜色融合的应用

并非所有源和目的因子结合都有意义，大多数应用只用到少量的结合方式。下面描述几种源和目的因子相结合的典型应用。例子中有些只用新的α 值，所以即使缓存中没有存储α 值也能执行。另外，经常存在多种方法来达到其中某些效果。

要绘制一幅由一个图像和另一个图像各占一半融合而成的图，实现的方法是：先将源因子设置为 GL_ONE，绘制第一个图像，然后将目的因子设置为 GL_SRC_ALPHA，将α 设置为 0.5，再绘制第 2 个图像。如果想由第一幅图像的 0.75 和第 2 幅图像的 0.25 来绘制一幅图像，则先绘制第一幅图像，再将α 设置为 0.25，并将调用 glBlendFunc 函数的参数分别设置为 GL_SRC_ALPHA 和 GL_ONE_MINUS_SRC_ALPHA。这是常用的一种颜色融合。

要等同地融合 3 种不同的图像，可以将源因子设置为 GL_SRC_ALPHA，目的因子设置为 GL_ONE，用α = 0.3333333 来绘制每一个图像，这样，每个图像的亮度只是其原始亮度的 1/3，明显可见，这些图像彼此是不重叠的。

假设正在编写一个绘图程序，需要有一支画笔能逐渐地在图像中增添内容，使每一笔划在图像的当前位置加入一定的颜色并与其融合(例如，每画一遍用 10%的颜色与 90%的背景图像融合)。要实现这一点，只要将画笔绘制的图像的α 设置为 0.1，并将 glBlendFunc 的参数设置为 GL_SRC_ALPHA 和 GL_ONE_MINUS_SRC_ALPHA 即可。

利用源或目的颜色融合函数(对于源因子为 GL_DST_COLOR 或 GL_ONE_MINUS_DST_COLOR，对目的因子为 GL_SRC_COLOR 或 GL_ONE_MINUS_SRC_COLOR)可以有效地调节每种颜色分量。这种操作等价于应用一个简单的滤色镜。例如，对红色分量乘以 60%，对绿分量乘以 40%，以及对蓝分量乘以 72%，可以模拟通过一个阻挡 40%的红光，60%的绿光和 28%的蓝光的滤色镜观察场景的效果。

假设需要绘制一幅有 3 个半透明面组成的图，有些面遮盖了另一些面，并且全部面在一个实心的背景上。如果最远的一个面能通过从它后面发出的颜色的 80%，中间的面能通过 40%，最前面的面能通过 90%。要形成这样一幅图，首先要利用默认的源和目的因子绘制背景，将融合因子改变为 GL_SRC_ALPHA 和 GL_ONE_MINUS_SRC_ALPHA，然后将α 设置为 0.2 绘制最远的面，将α 设置为 0.6 绘制中间的面，最后将α 设置为 0.1 绘制最前面的面。

12.1.3 一个颜色融合的例子

下面的程序绘制 4 个重叠的着色矩形，α 值都设置为 0.75，窗口的左下和右上象限被覆盖两次。在这两个象限中，利用源和目的融合因子 GL_SRC_ALPHA 和 GL_ONE_MINUS_SRC_ALPHA 按不同的次序融合颜色。在左下象限中，蓝绿色与原来的黄色相融合，在右上象限中，黄色与原来的青色相融合。其他象限用非融合颜色绘制。

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

```
#include <GL/glaux.h>

void myinit(void)
{
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glShadeModel(GL_FLAT);
    glClearColor(0.0, 0.0, 0.0, 0.0);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glColor4f(1.0, 1.0, 0.0, 0.75);
    glRectf(0.0, 0.0, 0.5, 1.0);

    glColor4f(0.0, 1.0, 1.0, 0.75);
    glRectf(0.0, 0.0, 1.0, 0.5);

/* 在右上角绘制反向的彩色多边形 */
    glColor4f (0.0, 1.0, 1.0, 0.75);
    glRectf (0.5, 0.5, 1.0, 1.0);

    glColor4f (1.0, 1.0, 0.0, 0.75);
    glRectf (0.5, 0.5, 1.0, 1.0);

    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D (0.0, 1.0, 0.0, 1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D (0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0, 1.0);
```

```
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

从上例可知，绘制矩形的次序影响最后的颜色。在左下象限中，蓝绿色矩形是源片段值，它与已在帧缓存中的黄色矩形相融合，黄色矩形为目的值。在右下象限中，黄色矩形是源，而青色矩形是目的。由于α 值全部为 0.75，所以实际的融合因子对于源来说为 0.75，对于目的来说都为 0.25。换句话说，源矩形有点半透明，但它比目的矩形更影响最后的颜色，因此在左下象限为淡青色，在右上象限为淡黄色。进行算术运算得到，左下象限 RGB 颜色为(0.25，1.0，0.75)，右上象限颜色为(0.75，1.0，0.25)。

12.1.4 利用深度缓存的三维融合

由 12.1.3 中的例子可知，绘制多边形的次序对颜色融合的结果影响很大。当绘制三维半透明物体时，从背面到正面和从正面到背面不同顺序绘制会得到不同结果。在确定适当的次序后，还需要考虑深度缓存。深度缓存(有时称为 Z 缓存)通常用于隐藏面消除。这个缓存保持对视点和屏幕窗口中占有特定像素的物体部分之间距离的跟踪，当另一种颜色在该像素处绘制时，只绘制最靠近视点的像素，此时，其深度值存储在深度缓存中。利用这种方法，表面的被遮挡部分不会被绘制出来，因此也不必用融合。

一般说来，同一场景往往需要同时绘制不透明和透明物体，并且需要利用深度缓存来对位于不透明物体后面的物体进行隐藏面消除。当一个不透明物体遮挡一个半透明物体或另一个不透明物体时，需要深度缓存消除较远距离的物体。但是，当半透明物体较近时，就需要将它与不透明物体融合。假如场景中任何物体都处于静止状态，则比较容易弄清多边形绘制的正确次序，但当视点或物体正在移动时，则比较困难。

解决办法是激活深度缓存。在绘制半透明物体时，使深度缓存工作于只读方式。首先，以正常操作方式利用深度缓存绘制所有不透明物体，然后，通过让深度缓存工作于只读方式来保留这些深度值。当绘制半透明物体时，它们的深度值要和不透明物体相比较，保证当它们在不透明物体后面时不被绘制出来。但当透明物体较靠近视点时，不会消除不透明物体，因为此时深度缓存是只读的，只考虑如何把它们与不透明物体融合起来。要控制深度缓存是否可写，可以利用 DepthMask()函数，参数设置为 GL_FALSE 表示设置缓存为只读，参数设置为 GL_TRUE 时，缓存可以进行正常的读写操作。

下面的程序利用上述方法在同一场景中绘制不透明物体和半透明物体。在这个程序中，单击调用 toggleviewpoint()改变视点位置，从而改变一个不透明的圆环和半透明圆环的次序。

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void)
{
    GLfloat mat_ambient[] = { 0.0,  0.0,  0.0,  0.15 };
    GLfloat mat_specular[] = { 1.0,  1.0,  1.0,  0.15 };
    GLfloat mat_shininess[] = { 15.0 };

    glMaterialfv(GL_FRONT,  GL_AMBIENT,  mat_ambient);
    glMaterialfv(GL_FRONT,  GL_SPECULAR,  mat_specular);
    glMaterialfv(GL_FRONT,  GL_SHININESS,  mat_shininess);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}

GLboolean eyePosition = GL_FALSE;

void CALLBACK toggleEye(AUX_EVENTREC *event)
{
    if (eyePosition)
        eyePosition = GL_FALSE;
    else
        eyePosition = GL_TRUE;
}

void CALLBACK display(void)
{
    GLfloat position[] = { 0.0,  0.0,  1.0,  1.0 };
    GLfloat mat_torus[] = { 0.75,  0.75,  0.0,  1.0 };
    GLfloat mat_cylinder[] = { 0.0,  0.75,  0.75,  0.15 };
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glPushMatrix();
       if (eyePosition)
          gluLookAt(0.0, 0.0, 9.0, 0.0, 0.0, 0.0, 0.0,
                    1.0, 0.0);
       else
          gluLookAt(0.0, 0.0, -9.0, 0.0, 0.0, 0.0, 0.0,
                    1.0, 0.0);
       glPushMatrix();
          glTranslatef(0.0, 0.0, 1.0);
          glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_torus);
          auxSolidTorus(0.275, 0.85);
       glPopMatrix();

       glEnable(GL_BLEND);
       glDepthMask(GL_FALSE);
       glBlendFunc(GL_SRC_ALPHA, GL_ONE);
       glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cylinder);
       glTranslatef(0.0, 0.0, -1.0);
       auxSolidCylinder(1.0, 2.0);
       glDepthMask(GL_TRUE);
       glDisable(GL_BLEND);
    glPopMatrix();

    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
int main(int argc, char** argv)
{
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition(0, 0, 500, 500);
    auxInitWindow(argv[0]);
    auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN, toggleEye);
    myInit();
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
}
```

下面再举一个较复杂的例子，在此例中，既包含半透明物体又包含不透明物体，而且既有不透明物体遮挡半透明物体和不透明物体的情况，也有半透明物体在两个不透明物体前面的情况，并且每个物体之间都是部分遮挡关系。

```
#include <windows.h>
#include <gl/glaux.h>
#include <GL/gl.h>
#include <GL/glu.h>

void DrawScene(void)
{
    // 绘制铅笔
    GLUquadricObj *obj = gluNewQuadric();
    gluQuadricDrawStyle(obj, GLU_FILL);
    gluQuadricNormals(obj, GLU_SMOOTH);
    gluQuadricOrientation(obj, GLU_OUTSIDE);

    // 绘制铅笔头
    glTranslated(25, -85, -400);
    gluLookAt(0.0, 70.0, 100, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glPushMatrix();
    glColor3f(0.2, 0.2, 0.2);
    glRotated(-90, 1, 0, 0);
    glRotated(-15, 0, 1, 0);
    gluCylinder(obj, 0, 4, 8, 50, 10);
    glTranslated(0, 0, 8);
    glColor3f(0.8, 0.9, 0.5);
    gluCylinder(obj, 4, 10, 12, 50, 10);
    // 绘制铅笔的其余部分
```

```
    glTranslated(0, 0, 12);
    glColor3f(1.0, 1.0, 0.0);
    gluCylinder(obj, 10, 10, 190, 8, 20);
    // 绘制铅笔的金属尾部和橡皮
    glTranslated(0, 0, 190);
    glColor3f(0.5, 0.5, 0.5);
    gluCylinder(obj, 10, 10, 20, 10, 10);
    glTranslated(0, 0, 20);
    glColor3f(1.0, 0.5, 0.5);
    gluCylinder(obj, 10, 10, 10, 20, 10);
    glPopMatrix();
    glPushMatrix();
    glTranslatef(-50, 50, -200);
    glColor3f(1.0, 0.0, 0.0);

    auxSolidSphere(60);
    glPopMatrix();

    // 绘制茶杯
    // 设置透明度
    glPushMatrix();
    glEnable(GL_BLEND);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glColor4f(0.2, 0.8, 0.2, 0.2);
    glRotated(-90, 1, 0, 0);
    gluCylinder(obj, 40, 80, 150, 50, 50);
    glDisable(GL_BLEND);
    glDepthMask(GL_TRUE);
    glPopMatrix();

}

void myinit (void)
{
    // 指定物体的初始材质、光照和位置
    GLfloat mat_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat mat_shininess[] = {100.0f};
    GLfloat light_ambient[] = {0.0f, 0.0f, 0.0f, 1.0f};
```

```
    GLfloat light_diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_position[] = {30.0f, 40.0f, -100.0f, 1.0f};

    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);

    glShadeModel(GL_SMOOTH);

    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);

    // 设置背景色
    glClearColor(0.3f, 0.3f, 0.3f, 0.0f);
}

GLvoid CALLBACK MyDisplay(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();

    DrawScene();
    glFlush();
    auxSwapBuffers();
}

void CALLBACK MyReshape(int w, int h)
{
```

```
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0f, (GLfloat) w/(GLfloat) h, 1.0f, 700.0f);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_DOUBLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 800, 700);
    auxInitWindow ("茶杯里的铅笔");
    myinit ();
    auxReshapeFunc (MyReshape);
    auxIdleFunc(MyDisplay);
    auxMainLoop(MyDisplay);
    return 0;
}
```

程序运行结果如图 12.1 所示。

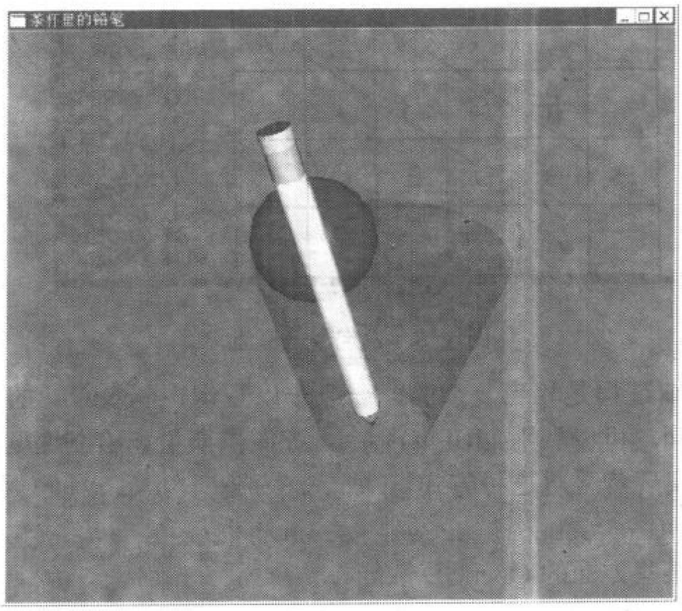

图12.1 插在杯子里的铅笔和杯子后面的球

12.2 反走样

如果不用附加的方法，在用 OpenGL 绘制直线时，特别是在绘制接近水平或接近垂直

的直线时，会出现很明显的锯齿状。出现锯齿的原因是由于屏幕上的像素是一系列小方格，任何图形都是由这些离散的小方格来近似模拟的。这种锯齿状在图形学中称为走样。下面介绍的反走样技术就是为了减少这种走样效果。图 12.2 表示走样的和反走样的两条交叉线。

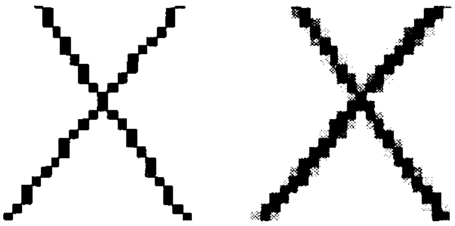

图12.2　走样和反走样直线

图 12.3 表示了一个像素宽的斜线是如何覆盖较多的像素方格的。当执行反走样时，OpenGL 根据每个片段值覆盖屏幕上的像素方格的覆盖百分比，计算每个片段值的覆盖率。图 12.3 右边表示这条直线的覆盖率值。在 RGBA 模式下，OpenGL 对片段值中α 值乘以覆盖率值，然后利用所得到的α 值把片段值与帧缓存中的相应像素融合起来。在颜色索引模式下，OpenGL 根据片段值的覆盖率，设置颜色索引的低 4 位(没有覆盖时为 0000，完全覆盖时为 1111)。直到装入查色表，并把查色表合理地利用于这些覆盖率信息。

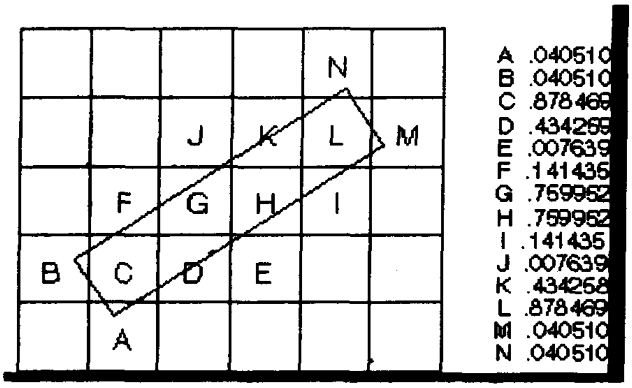

图12.3　确定覆盖率值

计算覆盖率的详细过程是相当复杂的，一般难以规定，OpenGL 的特定实现不同，计算覆盖率的过程也不同。可以利用 glHint()命令，在图像质量和绘制速度之间取一个折中，但并非所有 OpenGL 实现都采用这种方式。

```
void glHint(GLEnum target, Glenum hint);
```

此命令用以控制某些方面的 OpenGL 性能。

参数 target 指定一种符号常数，指出要控制的性能；hint 指定一种符号常数，指出所要求的性能。target 参数的取值范围及其含义如表 12.2 所示，hint 参数的取值范围及其含义如表 12.3 所示。

表12.2　target参数的取值范围及其含义

target 参数	含　义
GL_POINT_SMOOTH_HINT ， GL_LINE_SMOOTH_HINT，GL_POLYGON_SMOOTH_HINT	在执行反走样操作时，指定点、线或多边形所要求的采样质量
GL_FOG_HINT	指定雾的计算是逐个像素执行(GL_NICEST)还是逐个顶点执行(GL_FASTEST)
GL_PERSPECTIVE_CORRECTION_HINT	指定颜色和纹理坐标内插所要求的质量

表12.3　hint参数的取值范围及其含义

hint 参数	含　义
GL_FASTEST	指出选择最高效的选项
GL_NICEST	指出最精确，或最高质量的选项
GL_DINE_CARE	指出不选择

12.2.1　反走样点或线

要反走样点或线，需要用 glEnable()命令激活反走样，相应参数设置为 GL_POINT_SMOOTH 或 GL_LINE_SMOOTH。另外，还可能需要用 glHint 给出质量提示。下面分别按 RGBA 模式和颜色索引模式来描述这个处理过程。

1.　RGBA模式

在 RGBA 模式下，需要激活融合。大多数需要用到的融合因子是 GL_SRC_ALPHA(源)和 GL_ONE_MINUS_SRC_ALPHA(目的)。另外目的因子可以为 GL_ONE，以使直线在相交处稍微淡一些。先准备好要绘制的任意点或线。如果利用一个相当高的α 值，则由于执行融合必须考虑到绘制的次序，从而使反走样的效果显著。但是在没有出现严重与实际相反的情况时，可以不考虑绘制的次序。下面的程序例子设置了必要的方式对反走样进行初始化，然后绘制一个线框二十面体。注意，在本例中，没有激活深度缓存。

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void)
{
    GLfloat values[2];
    glGetFloatv(GL_LINE_WIDTH_GRANULARITY,  values);
```

```
    printf("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n",
              values[0]);

    glGetFloatv(GL_LINE_WIDTH_RANGE, values);
    printf("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n",
              values[0], values[1]);

    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth(1.5);

    glShadeModel(GL_FLAT);
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor4f(0.0, 0.0, 0.0, 1.0);
    auxWireIcosahedron(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -4.0);
}
```

```
void main(int argc, char** argv)
{
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition(0, 0, 400, 400);
    auxInitWindow(argv[0]);
    myinit();
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
}
```

2. 颜色索引方式

在颜色索引方式下实现反走样的困难在于查色表的装入和使用。由于颜色索引的最后 4 位给出了覆盖率值，所以必须利用从背景色到物体颜色的 16 个相邻的索引装入到索引表中。然后，用梯度表中的 16 种颜色的第一个值清除颜色缓存，用梯度表中的颜色来绘制点或直线。下例表明如何生成颜色梯度表，在颜色索引方式下绘制一个反走样的线框 20 面体。在本例中，颜色梯度表以索引 32 开始，装有灰度值。

在颜色索引方式下利用反走样的程序：

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

#define RAMPSIZE 16
#define RAMPSTART 32

void myinit(void)
{
    int i;

    for (i = 0; i < RAMPSIZE; i++) {
        GLfloat shade;
        shade = (GLfloat) i/(GLfloat) RAMPSIZE;
        auxSetOneColor(RAMPSTART+(GLint)i, shade, shade,
            shade);
    }

    glEnable (GL_LINE_SMOOTH);
    glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth (1.5);
```

```
    glClearIndex ((GLfloat) RAMPSTART);
    glShadeModel(GL_FLAT);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glIndexi(RAMPSTART);
    auxWireIcosahedron(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -4.0);
}

void main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_INDEX | AUX_DEPTH);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

由于颜色梯度表从背景色变化到物体颜色，所以当在背景上绘制物体时，反走样物体看上去是正确的，如果在另一个物体上绘制反走样物体时，则与物体相交的地方会产生不正确的颜色，除非已经构造了合适的颜色表。要获得最好的结果，可以利用深度缓存以确保像素颜色对应于最靠近视点的物体。在 RGBA 模式下，由于融合了两种物体的颜色，所

以看上去更自然。因此，当绘制由反走样的点和线组成的场景时，一般不用深度缓存。

12.2.2 反走样多边形

反走样填充多边形的边类似于反走样点和线。当不同的多边形具有重叠边时，必须适当地融合其颜色值，这既可以利用前面所描述的方法，也可以利用累积缓存对整个场景执行反走样。利用累积缓存的方法，虽然使用起来方便，但由于增加了很多计算量，所以执行起来很慢。

从理论上来说，反走样多边形既可以在 RGBA 模式下实现，也可以在颜色索引模式下实现，由于物体相交对多边形反走样的影响要比对点或线反走样影响大得多，因而绘制次序和融合精度很重要。当反走样不止一个多边形时，必须从正面到反面确定多边形的顺序，然后利用 glBlendFunc()，将源因子参数设置为 GL_SRC_ALPHA_SATURATE，将目的因子参数设置为 GL_ONE。因此，在颜色索引方式下实现反走样多边形通常是不实际的。

要在 RGBA 模式下实现反走样多边形，可以利用α 值表示多边形的覆盖率值，同时必须调用 glEnable(GL_POLYGON_SMOOTH)来激活多边形反走样。和直线反走样一样，需要根据多边形上的像素覆盖率来指定这些像素的部分α 值。另外，如果有必要，还可以设置 GL_POLYGON_SMOOTH_HINT 为其他值。

现在需要适当地融合重叠边。首先将深度缓存关闭，以便控制如何绘制重叠的像素，然后将融合因子设置为 GL_SRC_ALPHA_SATURATE 和 GL_ONE。通过这种特定的融合因子，最后的颜色值是源颜色和目的颜色的加权和。利用这种方法，可以使一个多边形边上一个像素与其后一个多边形上新绘制的颜色进行融合。最后，还需对场景中的所有多边形排序，从而在绘制之前使它们按从前到后的顺序排列好。

下面的例子描述如何对填充多边形进行反走样。单击可以激活或关闭反走样。注意，在例子中，进行绘制之前，已经裁剪了后向面，并且颜色缓存的α 值也被清为 0。程序如下：

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

GLboolean polySmooth;

void myinit(void)
{
   GLfloat mat_ambient[] = { 0.0,  0.0,  0.0,  1.00 };
   GLfloat mat_specular[] = { 1.0,  1.0,  1.0,  1.00 };
   GLfloat mat_shininess[] = { 15.0 };

   glMaterialfv(GL_FRONT,  GL_AMBIENT,  mat_ambient);
```

```
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glEnable (GL_LIGHTING);
    glEnable (GL_LIGHT0);
    glEnable (GL_BLEND);
    glCullFace (GL_BACK);
    glEnable (GL_CULL_FACE);
    glEnable (GL_POLYGON_SMOOTH);
    polySmooth = GL_TRUE;

    glClearColor (0.0, 0.0, 0.0, 0.0);
}

void CALLBACK toggleSmooth (AUX_EVENTREC *event)
{
    if (polySmooth) {
        polySmooth = GL_FALSE;
        glDisable (GL_BLEND);
        glDisable (GL_POLYGON_SMOOTH);
        glEnable (GL_DEPTH_TEST);
    }
    else {
        polySmooth = GL_TRUE;
        glEnable (GL_BLEND);
        glEnable (GL_POLYGON_SMOOTH);
        glDisable (GL_DEPTH_TEST);
    }
}

void CALLBACK display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.0, 0.0 };
    GLfloat mat_cube1[] = { 0.75, 0.75, 0.0, 1.0 };
    GLfloat mat_cube2[] = { 0.0, 0.75, 0.75, 1.0 };

    if (polySmooth)
        glClear (GL_COLOR_BUFFER_BIT);
    else
```

```
        glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix ();
        glTranslatef (0.0, 0.0, -8.0);
        glLightfv (GL_LIGHT0, GL_POSITION, position);

        glBlendFunc (GL_SRC_ALPHA_SATURATE, GL_ONE);

        glPushMatrix ();
            glRotatef (30.0, 1.0, 0.0, 0.0);
            glRotatef (60.0, 0.0, 1.0, 0.0);
            glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cube1);
            auxSolidCube (1.0);
        glPopMatrix ();

        glTranslatef (0.0, 0.0, -2.0);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cube2);
        glRotatef (30.0, 0.0, 1.0, 0.0);
        glRotatef (60.0, 1.0, 0.0, 0.0);
        auxSolidCube (1.0);

    glPopMatrix ();

    glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

void main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 200, 200);
```

```
    auxInitWindow (argv[0]);
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, toggleSmooth);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

12.3 雾 化

对于计算机绘制的图像和实际拍摄的图像，经常很容易就能分清。因为计算机绘制的图像的边缘往往太规则，显得很不真实。为了使人们觉得更真实许多方法，上一节中介绍的反走样技术对边缘进行平滑可以使绘制出的物体显得较真实。另外，还可以在整个场景中加入雾使物体随着远离而淡入，这样，物体就会显得更自然。这里所说的雾除了指真正的大气现象雾之外，还可指模拟的阴霾、薄雾、烟尘或污染物等。雾化效果在视觉模拟中是很重要的，因为常常需要模拟在一定能见度下的物体，尤其在飞行模拟中需要大量地利用雾化效果。

当雾化效果被激活后，离视点较远的物体开始淡入雾色中。可以通过控制雾的浓度来确定物体淡入远景的速率。在 RGBA 模式和颜色索引模式下均可利用雾化效果，只是在计算上稍有不同。由于雾化是在矩阵变换、光照和纹理映射之后进行的，所以雾对进行了变换、光照和纹理映射的物体同样起作用。同时，在一些模拟复杂场景程序中有时还可利用雾化来改善性能，因为浓雾后的物体往往可视为是不可见的，从而可以不对它们进行绘制。

12.3.1 雾的用法

雾的用法很简单，先调用 glEnable(GL_FOG)来激活雾，再利用 glFog*()来选择雾的颜色和控制雾的浓度方程即可。同时，还可以利用 glHint()来设置 GL_FOG_HINT 为某个值。下面的程序绘制 5 个红色的茶壶，每个茶壶离视点的距离不同。单击依次选择 3 种不同的雾方程。关于雾方程的定义将在 12.3.2 小节中详细讲解。

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>

GLint fogMode;

void CALLBACK cycleFog (AUX_EVENTREC *event)
{
```

```
    if (fogMode == GL_EXP) {
        fogMode = GL_EXP2;
        printf ("Fog mode is GL_EXP2\n");
    }
    else if (fogMode == GL_EXP2) {
        fogMode = GL_LINEAR;
        printf ("Fog mode is GL_LINEAR\n");
        glFogf (GL_FOG_START, 1.0);
        glFogf (GL_FOG_END, 5.0);
    }
    else if (fogMode == GL_LINEAR) {
        fogMode = GL_EXP;
        printf ("Fog mode is GL_EXP\n");
    }
    glFogi (GL_FOG_MODE, fogMode);
}

void myinit(void)
{
    GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };
    GLfloat local_view[] = { 0.0 };
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);
    glFrontFace (GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glEnable(GL_FOG);
    {
        GLfloat density;
        GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

        fogMode = GL_EXP;
        glFogi (GL_FOG_MODE, fogMode);
        glFogfv (GL_FOG_COLOR, fogColor);
        glFogf (GL_FOG_DENSITY, 0.35);
```

```
        glHint (GL_FOG_HINT, GL_DONT_CARE);
        glClearColor(0.5, 0.5, 0.5, 1.0);
    }
}
void renderRedTeapot (GLfloat x, GLfloat y, GLfloat z)
{
    float mat[3];
    glPushMatrix();
    glTranslatef (x, y, z);
    mat[0] = 0.113495; mat[1] = 0.01175; mat[2] = 0.01175;
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
    mat[0] = 0.61424; mat[1] = 0.0416617; mat[2] = 0.0416617;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
    mat[0] = 0.727811; mat[1] = 0.624220637; mat[2] = 0.624220637;
    glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
    glMaterialf (GL_FRONT, GL_SHININESS, 0.6*128.0);
    auxSolidTeapot(1.0);
    glPopMatrix();
}
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    renderRedTeapot (-4.0, -0.5, -1.0);
    renderRedTeapot (-2.0, -0.5, -2.0);
    renderRedTeapot (0.0, -0.5, -3.0);
    renderRedTeapot (2.0, -0.5, -4.0);
    renderRedTeapot (4.0, -0.5, -5.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= (h*3))
        glOrtho (-6.0, 6.0, -2.0*((GLfloat) h*3)/(GLfloat) w,
         2.0*((GLfloat) h*3)/(GLfloat) w, 0.0, 10.0);
    else
```

```
        glOrtho (-6.0*(GLfloat) w/((GLfloat) h*3),
         6.0*(GLfloat) w/((GLfloat) h*3), -2.0, 2.0, 0.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 450, 150);
    auxInitWindow (argv[0]);
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, cycleFog);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

12.3.2 雾方程

雾是利用雾的混合因子将雾与当前片段值颜色相融合产生的。因子 f 是下面 3 个方程中任意一个的计算结果，其结果被限定在[0，1]之间。

$f = e^{-(density \cdot z)}$ (GL_EXP)

$f = e^{-(density \cdot z)2}$ (GL_EXP2)

$f = \frac{end - z}{end - start}$ (GL_LINEAR)

其中 z 是视点和片断值中心之间的眼睛坐标距离。density，start 和 end 的值都是通过 glFog*()函数来指定的。因子 f 在 RGBA 模式和颜色索引模式下指定方法有所不同，下面加以说明。

```
void glFog{if}{v}(GLenum pname, TYPE param);
```

指定雾参数和雾方程。其中，pname 可以为以下符号常数之一：

- GL_FOG_MODE
- GL_FOG_DENSITY
- GL_FOG_START
- GL_FOG_END
- GL_FOG_INDEX
- GL_FOG_COLOR

param 用来指定 pname 要设置的值。

当 pname 设置为 GL_FOG_MODE 时，param 指定用于计算雾混合因子 f 的雾方程，param 的取值可以为以下 3 个符号常数之一：

GL_EXP(默认值)，GL_EXP2 和 GL_LINEAR。

- 当 pname 设置为 GL_FOG_DENSITY 时，param 指定 density(雾浓度)，默认值为 1.0。
- 当 pname 设置为 GL_FOG_START 时，param 指定 start (近距离)，默认值为 0.0。
- 当 pname 设置为 GL_FOG_END 时，param 指定 end(远距离)，默认值为 1.0。

在 RGBA 模式下，pname 可以设置为 GL_FOG_COLOR，在这种情况下 param 指向 RGBA 颜色的 4 个值。雾的默认颜色为(0.0, 0.0, 0.0, 0.0)。

在颜色索引模式下，pname 可以设置为 GL_FOG_INDEX，param 指定雾的颜色索引值，索引的默认值为 0。设置各种参数值绘出雾浓度方程的曲线，如图 12.4 所示。

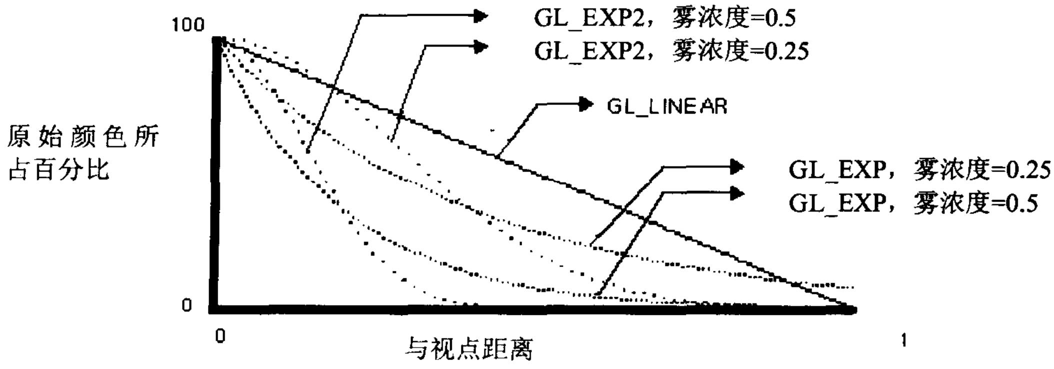

图12.4 雾浓度方程

在 RGBA 模式下，利用雾因子计算最后雾的颜色用下面的公式：

$$C = f\ Ci + (1 - f)\ Cf$$

其中，Ci 表示当前片断的 RGBA 值，Cf 表示由 GL_FOG_COLOR 所指定的雾的颜色值。

在颜色索引模式下，利用雾因子计算最后雾的颜色用下面的公式：

$$I = I_i + (1 - f)\ I_f$$

其中，I_i 表示当前片断的颜色索引，I_f 是由 GL_FOG_INDEX 所指定的雾的颜色索引。

要在颜色索引模式下利用雾，必须先在颜色梯度表中装入相应的值。梯度表中第一个颜色应为没有雾时物体的颜色，最后一个颜色应为物体完全被雾笼罩时的颜色。另外，还需要利用 glClearIndex()初始化背景的颜色索引，使它对应于梯度表中最后的颜色。这样，就把整个被雾笼罩的物体混合在背景中了。同样，在绘制物体之前，应调用 glIndex*()函数来指定颜色索引表中第一个颜色值，即未被雾笼罩的颜色。为了将雾应用于场景中不同的物体，必须建立几个颜色梯度表，并且在绘制每个物体之前调用 glIndex*()，设置当前颜色梯度表的起始索引。下面的程序例子说明如何初始化，然后在颜色索引模式下产生雾化效果。

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
```

```
#define NUMCOLORS 32
#define RAMPSTART 16

void myinit(void)
{
    int i;

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    for (i = 0; i < NUMCOLORS; i++) {
        GLfloat shade;
        shade = (GLfloat) (NUMCOLORS-i)/(GLfloat) NUMCOLORS;
        auxSetOneColor (16 + i, shade, shade, shade);
    }
    glEnable(GL_FOG);

    glFogi (GL_FOG_MODE, GL_LINEAR);
    glFogi (GL_FOG_INDEX, NUMCOLORS);
    glFogf (GL_FOG_START, 0.0);
    glFogf (GL_FOG_END, 4.0);
    glHint (GL_FOG_HINT, GL_NICEST);
    glClearIndex((GLfloat) (NUMCOLORS+RAMPSTART-1));
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
        glTranslatef (-1.0, -1.0, -1.0);
        glRotatef (-90.0, 1.0, 0.0, 0.0);
        glIndexi (RAMPSTART);
        auxSolidCone(1.0, 2.0);
    glPopMatrix ();

    glPushMatrix ();
        glTranslatef (0.0, -1.0, -2.25);
        glRotatef (-90.0, 1.0, 0.0, 0.0);
        glIndexi (RAMPSTART);
```

```
            auxSolidCone(1.0, 2.0);
        glPopMatrix ();
        glPushMatrix ();
            glTranslatef (1.0, -1.0, -3.5);
            glRotatef (-90.0, 1.0, 0.0, 0.0);
            glIndexi (RAMPSTART);
            auxSolidCone(1.0, 2.0);
        glPopMatrix ();
        glFlush();
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-2.0, 2.0, -2.0*(GLfloat)h/(GLfloat)w,
            2.0*(GLfloat)h/(GLfloat)w, 0.0, 10.0);
    else
        glOrtho (-2.0*(GLfloat)w/(GLfloat)h,
            2.0*(GLfloat)w/(GLfloat)h, -2.0, 2.0, 0.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_INDEX | AUX_DEPTH);
    auxInitPosition (0, 0, 200, 200);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

第 13 章　帧缓存技术与 OpenGL 动画

屏幕由像素的矩形阵列组成，每个像素能在图像中相应点显示一个小方格的颜色。要在屏幕上绘制这些像素，首先需要知道像素是什么颜色。颜色是存储在颜色缓存中的信息，全部像素的存储器称为缓存(buffer)。不同的缓存对每个像素包含有不同数目的数据，但在一个确定的缓存中，对每个像素指定相同数目的数据。存储像素单位个信息的缓存称为位面。

作为缓存的例子，现仔细地考察保存屏幕上显示颜色信息的颜色缓存，假如屏幕为 1280 像素宽和 1024 像素高，而且是全 2 位颜色屏幕，换句话说，可以显示 2 (或 16 777 216) 种不同的颜色，由于 24 位转换为 3 个字节(8 位/字节)，所以在本例中的颜色缓存对屏幕上 1024×1280(=1310720)个像素中的每个像素必须存储至少 3 个字节的数据。在物理屏幕上，特定的硬件系统可能有较多或较少的像素，而每个像素也可能有较多或较少的颜色数据。任何特定的颜色缓存，对屏幕上的每个像素保存相同数目的数据。

颜色缓存仅是保存像素信息的几种缓存中的一种。例如，保存每个像素的深度信息的是深度缓存。颜色缓存本身可以由几个子缓存组成，在一个系统上的帧缓存包含了所有这些缓存。除颜色缓存外，其他的一些缓存不能直接观察到，而是利用这些缓存来执行某些任务，如隐藏面消除、全场景的反走样、模板、绘平滑运动以及其他操作。

本章讲述在 OpenGL 实现中可以存在的全部缓存及其使用，其次讨论将任何数据写入到可显示的颜色缓存之前所执行的一系列检验和像素操作，最后，说明利用累积缓存如何累积颜色缓存中所绘制的图像。

13.1　缓存及其使用

OpenGL 系统可以操作如下一些缓存：

- 颜色缓存：前左、前右、后左、后右和任何数目的辅助颜色缓存。
- 深度缓存。
- 模板缓存。
- 累积缓存。

特定的 OpenGL 实现确定那些可用的缓存以及每个像素各保存多少位。另外，可以具有不同的可用缓存的多少个视屏(visuals)或窗口类型。至少保证有一个 RGBA 方式下使用的颜色缓存和具有非零大小颜色分量的关联的模板深度和累积缓存，以及一个颜色索引方式下使用的颜色缓存和关联的深度和模板缓存。表 13.1 列出了 glGetIntegerv()中有关查询 OpenGL 系统对特定视屏的每个像素的缓存存储所需要的参数。

表13.1　查询像素缓存位数的参数及其含义

参　数	含　义
GL_RED_BITS，GL_GREEN_BITS，GL_BLUE_BITS，GL_ALPHA_BITS	在颜色缓存中每个 R，G，B 或 A 分量的位数
GL_DEPTH_BITS	在深度缓存中每个像素的位数
GL_STENCIL_BITS	在模板缓存中每个像素的位数
GL_ACCUM_RED_BITS	在累积缓存中每个 R 分量的位数
GL_ACCUM_GREEN_BITS	在累积缓存中每个 G 分量的位数
GL_ACCUM_BLUE_BITS	在累积缓存中每个 B 分量的位数
GL_ACCUM_ALPHA_BITS	在累积缓存中每个 A 分量的位数

13.1.1　OpenGL 中的各种缓存

1.　颜色缓存

颜色缓存通常是绘图的缓存。它们包含颜色索引或 RGB 颜色数据，也可含有 A 值。OpenGL 实现支持立体(stereoscopic)视图，对左立体图和右立体图像有左和右颜色缓存。若不支持立体，只利用左缓存。同样，双缓存系统有前和后缓存，而单缓存系统只有前缓存。每个 OpenGL 实现必须提供一个前左颜色缓存。

此外，颜色缓存还支持最多 4 个任选的、不可显示的辅助颜色缓存。OpenGL 对这些缓存不指定任何特殊的使用方式，所以能够随意定义和使用它们。例如，可利用这些缓存来保存一个重复使用的图像，即把图像拷贝到通常的颜色缓存中，而不必重绘图像。利用 glCopyPixels()命令可进行图像的拷贝。

利用 glGetBooleanv()，设置 GL_STEREO 或 GL_DOUBLE_BUFFER 可查找系统是否支持立体(有左和右缓存)或双缓存(有前和后缓存)。要查询有多少辅助缓存，利用 glGetIntegerv()，设置 GL_AUX.BUFFERS。

2.　深度缓存

深度缓存存储每个像素的深度值。通常，用离眼睛的距离来度量深度，所以有较大深度缓存值的像素被较小值的像素所覆盖。深度缓存的状态可以通过“深度检验”进行修改。深度缓存有时称 z 缓存(z 来自这样的事实，x 和 y 值度量屏幕的水平和垂直于屏幕的距离)。

3.　模板缓存

模板缓存的一种用途是：把绘图局限于屏幕的某些部分，正如在一个喷涂罐头上，可以用一个卡纸(cardboard)模板作成相当精密的涂色图像。例如，要绘制一个通过畸形屏风出现的图像，可将屏风的图像存储在模板缓存中，然后绘制整个场景。模板缓存不绘出通过屏风未见到的任何东西。另外，如果应用是汽车驾驶模拟，就可对车内全部仪器及其他东西一次绘出，当汽车移动时，只需改变车外的场景。

4. **累积缓存**

如同 RGBA 方式下的颜色缓存作法一样，累积缓存保存 RGBA 颜色数据。累积缓存一般用于将一系列图像累积起来变成最后的、合成的图像(在颜色索引方式下利用累积缓存是无意义的)。利用这种方法，可以执行这样的操作：通过超采样图像，然后平均这些采样使其产生最后着色的值，变成颜色缓存的像素，以此执行场景反走样。绘图时不直接进入累积缓存。累积操作总是以矩形块执行，通常，数据经颜色缓存传进或传出。

13.1.2 清除缓存

在图形程序中，在 1280×1024 监视器上执行清除(或清除缓存中任一个)通常是最耗时的操作之一，这需要涉及到超过百万个像素的操作。对简单的图形应用，清除操作比其他绘图耗费更多的时间。譬如，当不仅需要清除颜色缓存，而且还要清除深度和模板缓存时，清除操作就要耗费 3 倍时间。

某些计算机解决这个问题是使硬件一次清除多个缓存。OpenGL 清除命令的构成就是利用这种结构。首先，指定要写到缓存中的清除值，然后发出一个简单命令执行清除操作，传送一个清除全部缓存的列表，如果硬件能够同时清除，则全部缓存一次性清除，否则顺序清除每个缓存。

如下命令可以分别对每个缓存设置清除值。

```
void glClearColor(GLCLampfred,  GLclampf green,  GLclampf blue,
         GLclampf alpha);
void glClearIndex(GLfloat index);
void glCIearDepthz(GLclampd depth);
void glCearStencil(GLint s);
void glCearAccum(GLfloat red, GLfloat green, GLfloat blue,  GLfloat
         alpha);
```

他们分别为指定颜色缓存(RGBA 方式)、颜色缓存(颜色索引方式)、深度缓存和累积缓存的当前清除值。GLclampf 和 GLclampd 类型的值置于 0.0 和 1.0 之间，默认的深度清除值为 1.0，其他的默认值为 0。清除命令设置的值在调用另一个相同命令改变值之前保持有效。

在选择清除之后和即将清除缓存时利用 glClear()

```
void glCear(GLbitfield mask);
```

清除指定的缓存。

mask 的值是 GL_COLOR_BUFFER_BIT，GL_DEPTH_BUFFER_BIT，GL_STENCIL_BUFFER_BIT 和 GL_ACCUM_BUFFER_BIT 按位逻辑 OR 的某种结合，以指明清除哪些缓存。GL_COLOR_BUFFER_BIT 清除 RGBA 颜色还是颜色索引缓存，取决于当时系统的方式。当清除颜色或颜色索引缓存时，清除允许修改的全部颜色缓存。

13.1.3 选择绘图的颜色缓存

绘图操作的结果可以进入任一个颜色缓存中，前、后、前左、后左、前右、后右，或任一个辅助缓存。可以选择单独的缓存作为绘图目标，也可设置缓存的结合作为绘图目标，例如，在双缓存应用中，要求在前、后缓存中绘制相同的背景，然后，要求只在后缓存中绘图，在完成绘图后交换缓存。但在有些情况下，要求处理特定的双缓存窗口中的部分，通过在前和后缓存中的绘图是单缓存处理。利用 glDrawBuffer ()命令选择绘图的缓存，即指定在哪些颜色结存中绘图。

```
void glDrawBuffer(GLenum mode);
```

mode 最多指定 4 种颜色缓存。mode 可设置如下符号常数中一个：

- GL_FRONT GL_FRONT_RIGHT GL_AUXI
- GL_BACK GL_FRONT_LEFT GL_FRONT_AND_BACK
- GL_RIGHT GL_BACK_RIGHT GL_NONE
- GL_LEFT GL_BACK_LEFT

对单缓存的现场，默认值为 GL_FRONT，对缓存的现场，默认值为 GL_BACK。

颜色被写到帧缓存，即写入到由 glDrawBuffer()所指定的颜色缓存中。以上符号常数说明如表 13.2 所示。

表13.2 mode参数的取值及其含义

取 值	含 义
GL_FRONT	仅写入前左和前右颜色缓存。当不存在前右颜色缓存时，只写入前左颜色缓存
GL_BACK	仅写入后左和后右颜色缓存。当不存在后右颜色缓存时，只写入后左颜色缓存
GL_RIGHT	仅写入前右和后右颜色缓存。当不存在后右颜色缓存时，只写入前右颜色缓存
GL_LEFT	仅写入前左和后左颜色缓存。当不存在后左颜色缓存时，只写入前左颜色缓存
GL_FRONT_RIGHT	仅写入前右颜色缓存
GL_FRONT_LEFT	仅写入前左颜色缓存
GL_BACK_RITHT	仅写入后右颜色缓存
GL_BACK_LEFT	仅写入后左颜色缓存
GL_AUXi	仅写入辅助颜色缓存 I
GL_FRONT.AND.BACK	写入全部前和后颜色缓存(前左、前右、后左、后右)。当不存在后颜色缓存进，仅写入前左和前颜色缓存。当不存在右颜色缓存时，仅写入前左和后左颜色缓存。当不存在右或后颜色缓存时，仅写入前左颜色缓存
GL_NONE	不写入颜色缓存

GL_AUXi 中的 i 表示特定辅助缓存的数字。通常总是：GL_AUXO+i。

13.1.4 屏蔽缓存

在 OpenGL 将数据写入到激活的颜色、深度，或模板缓存之前，利用下面命令对数据指定所用的屏蔽(masking)操作。用每个掩码执行按位逻辑 AND，并写入相应的数据。

```
void glIndexMask(GLuint mask);
void glColorMask(GLboolean red ,  GLboolean , GLboolean blue ,
        GLboolean alpha);
void glDepthMask(GLboolean flag);
void glStenclMask(GLuint mask);
```

设置用于控制写入所指定缓存的掩码。

利用 glIndexMask()设置的掩码仅适用于颜色索引方式。当 mask 中出现 1 时，写入颜色索引缓存中的相应值；当出现 0 时，不写入。同样，glColorMask()只对 RGBA 方式下绘图有效。red，green，blue 和 alpha 值控制写入相应的分量。(gl_TURE 含义为写相应分量。)当 glDepthMask()中 flag 设置为 GL_TRUE 时，激活写深度缓存，否则，关闭写。如同 glIndexMask()中用于颜色索引数据的 mask 一样，glStenciMask()中的 mask 用于模板数据。全部 Glboolean mask 的默认值为 GL_TRUE，以及对两个 Gluint mask 的默认值都为 1。

在颜色索引方式下的颜色屏蔽可以使用很多技巧。例如，可以利用索引中的每个位作为不同层，利用合适设置查色表来建立任意层之间的相互配合。用户可以建立上盖(overlays)和下衬(underlays)，以及作成通常所说的查色表动画。在 RGBA 方式下的屏蔽是很少用的，但可以利用它把个别的图像文件装入到红、绿和蓝位面中。

当对一系列帧要求有一个共同的背景，并需加上某些可能受到背景部分遮蔽的特性时，也可利用关闭写深度缓存。例如，假设背景为一片树林，设想绘一的树木重复帧，并有一些对象在树林中移动。在深度缓存中记录绘树的深度后，接着保存树的图像，用关闭写深度缓存来绘景物。只要新的景物不相互重叠，这幅图是正确的。要绘下一帧时，恢复树的图像，继续进行，不必恢复深度缓存中的值 。当背景极端复杂时，这种技巧是很有用的使复杂图像显示加快，这是由于重复拷贝图像于颜色缓存中要比重新计算几何形状快得多。

屏蔽该模板缓存可用一个多位模板缓存来保存多个模板(每位一个)。利用这种技术可进行模板检验中所解释的覆盖(capping)。

注意：*用 glStencilMask()指定的掩码来控制写模板的位面。这个掩码与 glStencilFunc()中第三个参量所指定的掩码是无关的，这个参数指定模板函数所考虑的那些方面。*

13.2 检验和操作片断值

当在屏幕上绘几何图形、正文、或图像时，OpenGL 执行旋转、平移、缩放某些等计算，确定光照，透视投影对象，算出窗口内哪些像素有作用和确定这些像素应绘什么颜色。

在本书第 6 章中给出有关如何控制这些操作的信息。在 OpenGL 确定生成各个片断值 和着什么色之后，有几个处理阶段仍继续控制是否如何和绘出作为进入帧缓存的像素的片断值 。例如，当片断值在窗口外，或离视点比帧缓存中的像素离视点较远时，就不绘出。在另一阶段中，片断值颜色与帧缓存中的像素颜色相融合。

本节描述在片断值进入帧缓存之前必须通过一系列的检验，以及当写片断值时可能对片断值执行最后的操作。检验和操作按以下顺序发生(假如片断值在先前检验中已被消除，则稍后的检验或操作就不再发生)：

- 剪裁检验
- a 检验
- 模板检验
- 深度检验
- 融合
- 抖动
- 逻辑操作

上列的检验和操作在下面分别详细描述。

13.2.1 剪裁检验

剪裁检验可定义窗口中的一个矩形部分。利用 glScissor()命令限制绘图发生在这个矩形内。当片断值在该矩形内时，即通过剪裁检验。

```
void glAlphaFunc(GLenum func, GLclampfref);
```

该命令设置剪裁矩形的位置和大小。

x，y 指定剪裁矩形的左下角。默认值为(0，0)。

width，height 指定剪裁矩形的宽和高。

位于剪裁矩形内的像素，通过剪裁检验。默认时，矩形区为窗口大小，并关闭剪裁。要激活和关闭剪裁，调用 glEnable()和 glDisable()，设置参数为 GL_SCISSOR。

剪裁检验是利用屏幕矩形区的模板检验的一种。在给定的系统利用模板时可能慢得多的情况下(或许因模板用软件执行的)，这种形式相当容易建立一种剪裁的最快硬件实现。

作为剪裁的一种高级应用，它可用于执行非线性投影。首先，把窗口剖分为规则的子区网格，指定视口和剪裁参数，限制一次绘制于一个区，而后利用不同的投影矩阵把整个场景投影于各个区。

调用 glIsEnabled()，设置 GL_SCISSOR_TEST，可确定剪裁是否激活。调用 glGetIntegerv()，设置 GL_SCISSOR_BOX，获取所定义的剪裁矩形的值。

13.2.2 a 检验

在 RGBA 方式下，a 检验允许根据 A 值来接受或排除片断值 。当激活时，a 检验比较到达的 A 值和参考值，接受或排除片断值，取决于比较的结果。利用 glAlphaFunc()同时设置参考值和比较函数。为了确定 a 检验是否激活，调 用 glIsEnabled()设置

GL_ALPHA_TEST；为了获取参考值　和比较函数，调用 glGetIntegerv()设置 GL_ALPHA_TEST_FUNC 或 GL_ALPHA_TEST_REF。

```
void glAlphaFunc(GLenum func, GLclampfref);
```

指定 a 检验的比较函数和参考值。

func　　指定 a 比较函数。func 的可能值及其含义在表 13.3 中列出。

ref　　指定参考值。ref 置于 0 和 1 之间。默认值为 0。

默认时比较函数为 GL_ALWAYS 同时关闭 a 检验。要激活和关闭 a 检验，调用 glEnable()和 glDisable()，设置 GL_ALPHA_TEST，有一种 a 检验的应用是实现透明算法。其做法是：绘制整个场景两次，第一次只接受 A 值为 1 的片断值，而第二次接受 A 值不等于 1 的片值，当两者都通过时接通深度缓存，当第二次通过时关闭写深度缓存。

另一种应用是利用纹理映射作成贴花，而且可以看透贴花的某些部分。设置贴花中的 A 值为 0.0，这是要求看透的部分；另外设置 A 值为 1.0，设置参考值为 0.5(或 0.0 和 1 .0 之间的任何值)，以及设置比较函数为 GL_GREATER。贴花看透的部分，在深度缓存中的值 是不受影响的。

13.2.3　模板检验

模板检验仅当存在模板缓存时发生。(当不存在模板缓存时，模板检验总是通过的。)模板利用的检验是：比较参考值和模板缓存中存储的值，根据检验的结果，修改模板缓存中的值。调用 glStencilFunc()和 glStencil0p()命令可以选择所用的特定比较函数、参考值，以及执行修改。

```
void glStencIFunc (Glenum func , Glintref , Gluint mask);
```

设置模板检验的比较函数、参考值和掩码。

func　指定检验的比较函数。有如下 8 种值：

GL_NEVER，GL_LESS，GL_LEQUAL，GL_GREATER，GL_GEQUAL，GL_EQUAL，GL_NOTEQUAL，GL_ALWAYS。

ref　指定模板检验的参考值。ref 置于[0，2n-1]范围内，这里 n 为模板缓存中的位面数。表 13.3 是 func 所接受的值：

表13.3　func的取值及其含义

func 取值	含　义
GL_NEVER	总是不通过
GL_LESS	当(ref & mask)<(stencil & mask)时通过
GL_LEQUAL	当(ref & mask)<(stencil &mask)时通过
GL_GREATER	当(ref &mask) > (stencil & mask)时通过
GL_GEQUAL	当(ref & mask)>(stencil & mask)时通过

续表13.3

func 取值	含　义
GL_EQUAL	当(ref & mask)=(stencil & mask) 时通过
GL_NOTEQUAL	当(ref & mask) =(stencil & make)时通过
GL_ALWAYS	总是通过

要激活和关闭模板检验，调用 glEnable()和 glDisable()，设置参数为 GL_STENCIL_TEST。默认情况下，func 为 GL-ALWAYS，ref 为 0，mask 全部为 1，并且模板检验关闭。

fail　　当模板检验失败时，指定采取的动作。接受以下 6 个符号常数：

GL_KEEP，GL_ZERO，GL_REPLACE，GL_INCR，GL_DECR，GL_INVERT。

zfail　　当模板检验失败时，且深度检验失败时，指定模板的动作。 zfail 接受与 fail 相同的符号常数。

zpass　　当模板检验和深度检验都通过时，或者当模板检验通过和不存在深度缓存或未激活深度时，指定模板的动作。zpass 接受与 fail 相同的符号常数。6 种可能的动作如表 13.4 所示。

表13.4　6种可能的动作常数及其含义

动作常数	含　义
GL_KEEP	保持当前值
GL_ZERO	设置模板缓存值为 0
GL_REPLACE	设置模板缓存的值为 glStencilFunc 指定的 ref
GL_INCR	递增当前模板缓存的值。趋于最大可表示的无符号整数(若模板缓存保持 s 位，则为 2s—)
GL_DECR	递减当前模板缓存的值。趋于 0
GL_INVERT	逐位倒置当前模板缓存的值

默认时，全部 3 个模板函数均为 GL_KEEP。

利用查询函数 glGetIntegerv()和表 13.5 列出的值中一个，可以获取全部 6 个与模板有关的参数值。此外，还能通过调用 glIsEnabled()，设置 GL_STENCIL_TEST 来确定模板检验是否激活。

表13.5　查询各种模板检验是否被激活

查 询 值	含　义
GL_STENCIL_FUNC	模板函数
GL_STENCIL_REF	模板参考值
GL_STENCIL_VALUE_MASK	模板掩码
GL_STENCIL_FAIL	模板失败时的动作

续表13.5

查询值	含义
GL_STENCIL_PASS_DEPTH_FAIL	模板通过和深度缓存失败的动作
GL_STENCIL_PASS_DEPTH_PASS	模板通过和深度缓存通过的动作

模板检验的最典型应用是将屏幕上规则形状的区域屏蔽掉，以使出现这个区域中的部分不绘出。要实现这种应用，先在模板掩码中填入口，而后在模板缓存中用 1 绘制所要求的内容。通常不能直接在模板缓存中绘图，但可以通过在颜色缓存中绘图和对 zpass 函数选择一个合适的值(例如 GL_REPLACE)，以达到相同的结果。无论何时进行图形绘制，都会同时在模板缓存中写入一个值(此时，作为参考值)。为了使模板缓存绘图不受到颜色缓存内容的影响，设置颜色掩码为 0(或 GL_FALSE)，此时，也要求关闭写到深度缓存中。

在定义模板区域后，设置参考值为 1，而且比较函数使参考值等于模板面的值时通过片断值。在绘图期间，不修改模板面的内容。

下面的程序表明如何用这种方法来实现模板检验。绘两个圆环面及场景中央有一个菱形的剪裁式样，在菱形模板掩内绘一个球体。在本例中，仅当重绘窗口时才发生进入模板缓存中绘图，所以在生成模板掩码后，即清除颜色缓存。

覆盖(capping)——的方法就是假设绘一个由几个多边形组成的闭合凸对象，并有一个与对象相交的裁剪面，将对象切一块，现需要用某种颜色的面覆盖对象，使见不到对象的内部。要这样做，应先清除模板缓存为 0，激活模板和设置模板比较函数为总是接受片断值　，开始绘图。每次接受片断值时，倒置模板面中的值。在绘出全部对象后，需要不覆盖的屏幕区在模板面中为 0，而需要覆盖的区域为非 0。重置模板函数只绘模板值为非零的地方，并且通过整个屏幕绘一个覆盖颜色的大多边形。

重叠半透明多边形，假设有一个由稍微重叠的多边形组成的半透明面。如果只利用 a 融合，则下衬对象的部分被多个透明面所覆盖，这样并不好看。利用模板面来确保每个片断值最多被透明面的一部分覆盖。具体做法是：清除模板面为 0，仅当模板面为 0 时绘图，并当绘图时递增模板面中的值　。

点画——假设需要绘一个具有点画模式的图像。具体做法是：把点画模式写到模板缓存中，而后根据缓存的存储信息有条件地绘图。在给出原始的点画模式后，绘图像时，模板缓存不改变，以便使模板面中的模式得到点画的对象。

```
// 使用模板检验来裁剪物体的简单示例程序
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <windows.h>
#include <GL/gl.h>

static char *winName = "Simple Capping";
static int winX = 0, winY = 0;
```

```
static int winWidth = 256, winHeight = 256;

void drawCylinder(void)
{
    const int numMajor = 8;
    const int numMinor = 32;
    const float height = 1.2F;
    const float radius = 0.6F;
    float majorStep = height / numMajor;
    float minorStep = 2.0F*3.14159F / numMinor;
    int i, j;

    for (i=0; i<numMajor; ++i) {
    GLfloat z0 = 0.5F*height - i * majorStep;
    GLfloat z1 = z0 - majorStep;

    glBegin(GL_TRIANGLE_STRIP);
    for (j=0; j<=numMinor; ++j) {
        double a = j * minorStep;
        GLfloat nx = (GLfloat) cos(a);
        GLfloat ny = (GLfloat) sin(a);
        GLfloat x = radius * nx;
        GLfloat y = radius * ny;

        glNormal3f(nx, ny, 0.0F);
        glVertex3f(x, y, z0);

        glNormal3f(nx, ny, 0.0F);
        glVertex3f(x, y, z1);
    }
    glEnd();
    }

    glNormal3f(0.0F, 0.0F, 1.0F);
    glBegin(GL_TRIANGLE_FAN);
    for (i=0; i<numMinor; ++i) {
    float a = i * minorStep;
    GLfloat x = radius * (GLfloat) cos(a);
    GLfloat y = radius * (GLfloat) sin(a);
```

```
        glVertex3f(x, y, 0.5F*height);
        }
        glEnd();

        glNormal3f(0.0F, 0.0F, -1.0F);
        glBegin(GL_TRIANGLE_FAN);
        for (i=0; i<numMinor; ++i) {
        float a = i * minorStep;
        GLfloat x = radius * (GLfloat) cos(a);
        GLfloat y = radius * (GLfloat) sin(a);

        glVertex3f(-x, y, -0.5F*height);
        }
        glEnd();
}

void setMaterial(void)
{
    GLfloat ambient[4] = { 0.05F, 0.05F, 0.05F, 1.0F };
    GLfloat diffuse[4] = { 0.6F, 0.0F, 0.2F, 1.0F };
    GLfloat specular[4] = { 1.0F, 1.0F, 1.0F, 1.0F };

    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, diffuse);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 20.0F);
}

void initialize(void)
{
    /* 设置投影变换 */
    glMatrixMode(GL_PROJECTION);
    glFrustum(-0.5F, 0.5F, -0.5F, 0.5F, 1.0F, 3.0F);

    /* 设置视图变换 */
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.0F, 0.0F, -2.0F);
```

```
    glClear(GL_STENCIL_BUFFER_BIT);

    glEnable(GL_DEPTH_TEST);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    setMaterial();
}

void resize(void)
{
    /* 设置视口使之覆盖整个窗口 */
    glViewport(0, 0, winWidth, winHeight);

    glClear(GL_STENCIL_BUFFER_BIT);
}

GLboolean moveObject;
GLfloat obX = 45.0F, obY = 45.0F, obZ;

GLboolean moveClipPlane;
GLfloat cpX = -45.0F, cpY = -45.0F, cpZ;

GLdouble clipPlane[4] = { 0.0, 0.0, -1.0, 0.5 };

GLfloat clipPlaneNormal[3] = {
    0.0F, 0.0F, 1.0F,
};
GLfloat clipPlaneVerts[4][3] = {
    { 10.0F, 10.0F, 0.5F },
    {-10.0F, 10.0F, 0.5F },
    {-10.0F, -10.0F, 0.5F },
    { 10.0F, 10.0F, 0.5F },
};

void setClipPlane(void)
{
    glPushMatrix();
```

```
    glRotatef(cpX, 1.0F, 0.0F, 0.0F);
    glRotatef(cpY, 0.0F, 1.0F, 0.0F);
    glRotatef(cpZ, 0.0F, 0.0F, 1.0F);

    glClipPlane(GL_CLIP_PLANE0, clipPlane);

    glPopMatrix();
}

void drawClipPlane(void)
{
    glPushMatrix();
    glRotatef(cpX, 1.0F, 0.0F, 0.0F);
    glRotatef(cpY, 0.0F, 1.0F, 0.0F);
    glRotatef(cpZ, 0.0F, 0.0F, 1.0F);

    glNormal3fv(clipPlaneNormal);
    glBegin(GL_QUADS);
    glVertex3fv(clipPlaneVerts[0]);
    glVertex3fv(clipPlaneVerts[1]);
    glVertex3fv(clipPlaneVerts[2]);
    glVertex3fv(clipPlaneVerts[3]);
    glEnd();

    glPopMatrix();
}

void redraw(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* 物体模型变换 */
    glPushMatrix();
    glRotatef(obX, 1.0F, 0.0F, 0.0F);
    glRotatef(obY, 0.0F, 1.0F, 0.0F);
    glRotatef(obZ, 0.0F, 0.0F, 1.0F);

    /* 相对于物体定位裁剪面 */
    setClipPlane();
```

```
    glEnable(GL_STENCIL_TEST);

    /*
    ** 使用模板检验函数使物体里面的内容露出来
    */
    glStencilFunc(GL_ALWAYS, 1, 1);
    glStencilOp(GL_KEEP, GL_INVERT, GL_INVERT);

    glEnable(GL_CLIP_PLANE0);

    drawCylinder();

    glDisable(GL_CLIP_PLANE0);

    /*
    ** 中有当(vef &mask)=(stencil&mask)时才允许模板检验通过。同时设置模
           板缓存值为 0，从而在绘制中。每一帧画面时就不需要再清除模板缓存。
    */
    glStencilFunc(GL_EQUAL, 1, 1);
    glStencilOp(GL_KEEP, GL_ZERO, GL_ZERO);

    drawClipPlane();

    glDisable(GL_STENCIL_TEST);
    glPopMatrix();
}

static char *className = "OpenGL";
static HDC hDC;
static HGLRC hGLRC;
static HPALETTE hPalette;
static BOOL needsRedisplay = TRUE;

void doRedisplay(void)
{
    if (needsRedisplay) {
    redraw();
    SwapBuffers(hDC);
```

```
    needsRedisplay = FALSE;
    }
}

void redisplay(void)
{
    needsRedisplay = TRUE;
}

void setupPixelFormat(HDC hDC)
{
    PIXELFORMATDESCRIPTOR pfd = {
    sizeof(PIXELFORMATDESCRIPTOR),      /* 结构大小 */
    1,                 /* 版本 */
    PFD_SUPPORT_OPENGL |
    PFD_DRAW_TO_WINDOW |
    PFD_DOUBLEBUFFER,           /* 支持双缓存 */
    PFD_TYPE_RGBA,              /* 颜色模式 */
    16,                 /* 颜色深度 */
    0, 0, 0, 0, 0, 0,           /* 颜色位(可忽略) */
    0,                 /* 没有非透明度缓存 */
    0,                 /* 忽略非透明度位 */
    0,                 /* 没有累加缓存 */
    0, 0, 0, 0,             /* 累加位 (忽略) */
    16,                 /* 深度缓存 */
    1,                 /* 没有模板缓存 */
    0,                 /* 没有辅助缓存 */
    PFD_MAIN_PLANE,             /* 主层 */
    0,                 /* 保留 */
    0, 0, 0,             /* 不设置层可见性和损毁掩模 */
    };
    int pixelFormat;

    pixelFormat = ChoosePixelFormat(hDC, &pfd);
    if (pixelFormat == 0) {
    MessageBox(WindowFromDC(hDC), "ChoosePixelFormat failed.",
            "Error",
      MB_ICONERROR | MB_OK);
    exit(1);
```

```
    }

    if (SetPixelFormat(hDC, pixelFormat, &pfd) != TRUE) {
    MessageBox(WindowFromDC(hDC) ,    "SetPixelFormat  failed." ,
            "Error",
       MB_ICONERROR | MB_OK);
    exit(1);
    }
}

void setupPalette(HDC hDC)
{
    int pixelFormat = GetPixelFormat(hDC);
    PIXELFORMATDESCRIPTOR pfd;
    LOGPALETTE* pPal;
    int paletteSize;

DescribePixelFormat(hDC, pixelFormat, sizeof (PIXELFORMATDESCR-
            IPTOR), &pfd);

    if (pfd.dwFlags & PFD_NEED_PALETTE) {
    paletteSize = 1 << pfd.cColorBits;
    } else {
    return;
    }

    pPal = (LOGPALETTE*)
malloc(sizeof(LOGPALETTE) + paletteSize * sizeof(PALETTEENTRY));
    pPal->palVersion = 0x300;
    pPal->palNumEntries = paletteSize;

    /* build a simple RGB color palette */
    {
    int redMask = (1 << pfd.cRedBits) . 1;
    int greenMask = (1 << pfd.cGreenBits) . 1;
    int blueMask = (1 << pfd.cBlueBits) . 1;
    int i;

    for (i=0; i<paletteSize; ++i) {
```

```
            pPal->palPalEntry[i].peRed =
                (((i >> pfd.cRedShift) & redMask) * 255) / redMask;
            pPal->palPalEntry[i].peGreen =
                (((i >> pfd.cGreenShift) & greenMask) * 255) / greenMask;
            pPal->palPalEntry[i].peBlue =
                (((i >> pfd.cBlueShift) & blueMask) * 255) / blueMask;
            pPal->palPalEntry[i].peFlags = 0;
        }
        }

        hPalette = CreatePalette(pPal);
        free(pPal);

        if (hPalette) {
        SelectPalette(hDC, hPalette, FALSE);
        RealizePalette(hDC);
        }
}
LRESULT APIENTRY WndProc(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    static GLint x0, x1, y0, y1;

    switch (message) {
    case WM_CREATE:
    return 0;
    case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
    case WM_SIZE:
    /* 跟踪窗口大小的变化 */
    if (hGLRC) {
        winWidth = (int) LOWORD(lParam);
        winHeight = (int) HIWORD(lParam);
        resize();
        return 0;
```

```
}
case WM_PALETTECHANGED:
/* 在非当前窗口的情况下实现调色板 */
if (hGLRC && hPalette && (HWND) wParam != hWnd) {
    UnrealizeObject(hPalette);
    SelectPalette(hDC, hPalette, FALSE);
    RealizePalette(hDC);
    redisplay();
    break;
}
break;
case WM_QUERYNEWPALETTE:
/* 在是当前窗口的情况下实现调色板 */
if (hGLRC && hPalette) {
    UnrealizeObject(hPalette);
    SelectPalette(hDC, hPalette, FALSE);
    RealizePalette(hDC);
    redisplay();
    return TRUE;
}
break;
case WM_PAINT:
{
    PAINTSTRUCT ps;
    BeginPaint(hWnd, &ps);
    if (hGLRC) {
    redisplay();
    }
    EndPaint(hWnd, &ps);
    return 0;
}
break;
case WM_CHAR:
/* 处理键盘输入 */
switch ((int)wParam) {
case VK_ESCAPE:
    DestroyWindow(hWnd);
    return 0;
case ' ':
```

```
        redisplay();
        break;
    default:
        break;
    }
    break;
    case WM_LBUTTONDOWN:
    if (!moveObject && !moveClipPlane) {
        SetCapture(hWnd);
        moveObject = GL_TRUE;
        x0 = ((int) LOWORD(lParam) << 16) >> 16;
        y0 = ((int) HIWORD(lParam) << 16) >> 16;
        redisplay();
    }
    break;
    case WM_RBUTTONDOWN:
    if (!moveObject && !moveClipPlane) {
        SetCapture(hWnd);
        moveClipPlane = GL_TRUE;
        x0 = ((int) LOWORD(lParam) << 16) >> 16;
        y0 = ((int) HIWORD(lParam) << 16) >> 16;
        redisplay();
    }
    break;
    case WM_LBUTTONUP:
    case WM_RBUTTONUP:
    if (moveObject || moveClipPlane) {
        ReleaseCapture();
        moveObject = moveClipPlane = FALSE;
    }
    break;
    case WM_MOUSEMOVE:
    x1 = ((int) LOWORD(lParam) << 16) >> 16;
    y1 = ((int) HIWORD(lParam) << 16) >> 16;
    if (moveObject) {
        obX += 360.0F * (y1 - y0) / winHeight;
        obY += 360.0F * (x1 - x0) / winWidth;
    } else if (moveClipPlane) {
        cpX += 360.0F * (y1 - y0) / winHeight;
```

```
            cpY += 360.0F * (x1 - x0) / winWidth;
        }
        x0 = x1;
        y0 = y1;
        redisplay();
        break;
        default:
        break;
        }
        return DefWindowProc(hWnd, message, wParam, lParam);
}

int APIENTRY WinMain(
        HINSTANCE hCurrentInst,
        HINSTANCE hPreviousInst,
        LPSTR lpszCmdLine,
        int nCmdShow)
{
        WNDCLASS wndClass;
        HWND hWnd;
        MSG msg;

        /* 注册窗口类 */
        wndClass.style = CS_HREDRAW | CS_VREDRAW;
        wndClass.lpfnWndProc = WndProc;
        wndClass.cbClsExtra = 0;
        wndClass.cbWndExtra = 0;
        wndClass.hInstance = hCurrentInst;
        wndClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndClass.hbrBackground = NULL;
        wndClass.lpszMenuName = NULL;
        wndClass.lpszClassName = className;
        RegisterClass(&wndClass);

        /* 创建窗口 */
        hWnd = CreateWindow(
        className, winName,
        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
```

```
winX, winY, winWidth, winHeight,
NULL, NULL, hCurrentInst, NULL);

/* 显示窗口 */
ShowWindow(hWnd, SW_SHOWNORMAL);
UpdateWindow(hWnd);

/* 为 OpenGL 绘制的初始化 */
hDC = GetDC(hWnd);
setupPixelFormat(hDC);
setupPalette(hDC);
hGLRC = wglCreateContext(hDC);
wglMakeCurrent(hDC, hGLRC);
initialize();

/* 处理消息 */
while (1) {
if (needsRedisplay &&
    PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == FALSE)
{
    doRedisplay();
}
if (GetMessage(&msg, NULL, 0, 0) != TRUE) {
    break;
}
TranslateMessage(&msg);
DispatchMessage(&msg);
doRedisplay();
}

/* 结束 OpenGL 绘制 */
wglMakeCurrent(NULL, NULL);
wglDeleteContext(hGLRC);
if (hPalette) {
DeleteObject(hPalette);
}
ReleaseDC(hWnd, hDC);

return msg.wParam;
```

```
}
```

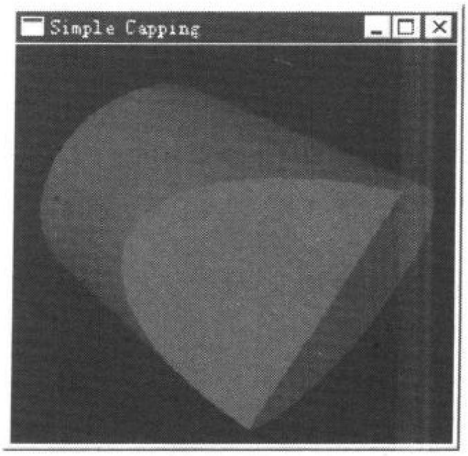

图13.2 简单模板检验程序运行结果

13.2.4 深度检验

对屏幕上的每个像素，深度缓存保持对视点和像素所在对象之间距离的跟踪。而后，当所指定的深度检验通过时，输入的深度值替换深度缓存中的值。

深度缓存通常用于隐藏面消除，如果像素出现一个新的候选颜色，则仅当相应对象较前面对象近才绘制。利用这种方法，只有当对象未被其他景物遮蔽时，才能在绘制整个场景后保留下来。在初始化时，通常将一个离视点尽可能远的值填入深度缓存中，以使任何对象都比这个值近。要利用深度缓存，只需调用 glEnable()，设置参数为 GL_DEPTH_TEST，来激活深度缓存。另外，在重绘每帧前要清除深度缓存。此外，还可利用 glDepthFunc()函数选择不同的比较函数，进行深度检验。

```
void glDepthFunc(GLenum func);
```

指定深度缓存比较所用的值。

func 用于指定深度比较函数。可接受的符号常数如表 13.6 所示。

表13.6 func参数可接受的符号常数及其含义

符号常数	含 义
GL_NEVER	不通过
GL_LESS	当输入的 z 值小于存储的 z 值时通过
GL_EQUAL	当输入的 z 值等于存储的 z 值时通过
GL_LEQUAL	当输入的 z 值小于或等于存储的 z 值时通过
GL_GREATER	当输入的 z 值大于存储的 z 值时通过
GL_NOTEQUAL	当输入的 z 值不等于存储的 z 值时通过
GL_GEQUAL	当输入的 z 值大于或等于存储的 z 值时通过
GL_ALWAYS	总是通过

13.2.5 融合、抖动和逻辑操作

一旦输入的片断值全部通过前面所描述的检验后，这个片断值就能用某一种方法与颜色缓存中当前的颜色存储信息相结合。最简单的默认方法是复写当前值。另一种方法，当利用 RGBA 方式及要求片断值用于半透明或反走样时，可将该值与缓存中的值平均(融合)。在少量可用颜色的系统中，可要求抖动颜色值，以损失分辨率的代价来增加可用颜色的数目。最后，在颜色索引方式下，可以利用任意的逐位逻辑操作来结合输入的片断值和已写的像素。

1. 融合

融合结合输入的片断值 R、G、B 和 A 值和在该位置上已存储的像素值。融合可以应用不同的融合操作，而且融合的发生依赖于输入的 A 值和该像素存储的 A 值(如存在的话)。

2. 抖动

在少量颜色位面系统上，可以通过抖动(dithering)图像中的颜色，以空间分辨率为代价来改进颜色分辨率。抖动类似报纸上的半色调(halftoning)。虽然报纸上只有两种颜色(黑和白)，但能用黑、白点的结合，以灰度表示照片。比较新闻图片(没有灰度)和原始照片(有灰度)，造成空间分辨率的损失是明显的。由于即使最低质量的颜色显示通常也至少有几种不同的红、绿和蓝可用值(不像报纸上只有两种)，所以在空间分辨方面以比较小的损失来换取一个较好的颜色范围。

抖动操作的发生是与硬件有关的。全部 OpenGL 允许接通或关闭抖动操作，在某些机器上，激活抖动可能完全没有作用，这是由于机器已经具有很高的颜色分辨率。要激活和关闭抖动，调用 glEnable()和 glDisable()，设置 GL_DITHER。默认时，抖动是激活的。

抖动适用于 RGBA 颜色和颜色索引 2 种方式。颜色和颜色索引在两种最接近的可能性之间，以某种与硬件有关的方法交替进行。例如，在颜色索引方式下，如果激活抖动且着色的颜色索引为 4.4，则像素的 6/10 用索引 4 着色，另外的 4/10 用索引 5 着色。在 RGBA 方式下，抖动是对每个分量(包括 A)分别执行的。要在颜色索引方式下利用抖动，一般需要把查色表中的颜色按梯度适当排列，不然可能得到奇怪的图像。

在 RGBA 方式下，抖动是把最后值写到颜色缓存中之前的最后步骤。在颜色索引方式下，可以执行下面描述的逻辑操作中的一个。

3. 逻辑操作

在颜色索引方式下，颜色索引可用整数位模式说明。对明暗处理和抖动，通常用整数说明较好。但对结合不同层上绘图所组成的图像(例如，当利用写掩码来限制一组不同的位面绘图时)，用位模式说明更好。逻辑操作(例如 OR 或 XOR)适用于输入的片断值和/或颜色缓存中当前值。

这些片断值的操作在 bitblt 型的机器上是很有用的。在这类机器上，主要的图形操作是把矩形数据块从窗口中一个地方拷贝到另一个地方，从窗口拷贝到存储器，或从存储器

拷贝到窗口。一般来说，这种拷贝并不把数据直接写到存储器中，而是允许对输入的数据和已有的数据执行任意的逻辑操作，然后用操作的结果替换现有的数据。由于这个过程用硬件可以很方便地实现，所以使用这类机器是很有效的。举一个利用逻辑操作的例子，XOR 可用于绘一种可恢复原状的图像，只要对同一绘图执行 XOR，即恢复原始图像。

利用 glLogicOp()在 16 个逻辑操作中选择。要激活和关闭逻辑操作，调用 glEnable() 和 glDisable()，并设置参数为 GL_LOGIC_OP。

13.3 累积缓存

累积缓存可应用于如场景反走样、运动模糊、模拟逼真的视野深度，以及计算由多个光源形成的软阴影等。特别与一些其他缓存相结合的技术也是可能的。

OpenGL 的图形操作不直接写到累积缓存中。通常，在某个标准颜色缓存中生成一系列图像，并且把这些图像每次累加到累积缓存中。当完成累加后，把结果转到颜色缓存中加以显示。为了减少舍入误差，累积缓存是具有比标准颜色缓存更高的精度(每种颜色有较多位)。绘制场景几次显然比绘制一次要耗用较多时间，但质量较高。可以采用一种适合于应用的质量和绘制时间之间的折中方案。

可以像摄影师对胶卷多次曝光的方法一样利用累积缓存。通常，摄影师通过不推进胶卷来拍摄几张同一场景的照片，当场景中的任何东西移动时，对象就出现模糊。计算机绘制图像比摄影师用照相机能做更多的事情，例如，计算机可以灵活控制遍及的视点，但摄影师摇动照相机却不能做到可预估和可控制的程度。

可以利用 glAccum()函数控制累积缓存。

```
void glAccum(GLenum op, GLfloat, value);
```

op　指定累积缓存的操作。接受如下符号常数：

GL_ACCUM，　GL_LOAD，　GL_ADD，　GL_MULT 和 GL_RETURN。

value 指定累积缓存操作中所用的浮点值。op 确定如何利用 value。

op 设置的符号常数含义如下：

- GL_ACCUM　从当前选择的读缓存中(参阅第 8 章中 glReadBuffer())读出每个像素的 R，G，B 和 A 值，并乘以 value 并加上累积缓存中相应的像素分量。结果为[0，1]范围中的浮点值 。
- GL_LOAD　从当前选择的缓存中读 R，G，B 和 A 值，并乘以 value，而后存储于相应累积缓存单元中，复写当前值。
- GL_ADD　将 value 加上累积缓存中每个 R，G，B 和 A。
- GL_MULT　累积缓存中每个 R，G，B 和 A 乘以 value。
- GL_ADD 和 GL_MULT 的结果都返回到累积缓存中。对 GL_MULT，value 置于[-1.0，1.0]范围内。对 GL_ADD，无此限制。
- GL_RETURN　将累积缓存中的值传送到当前选择的写颜色缓存中。每个 R，G，B 和 A 分量乘以 value ，而后存储于相应的颜色缓存单元中。

为了执行场景反走样，首先清除累积缓存和激活读和写的前缓存，而后通过以稍微不同的位置绘图像的程序循环几次(譬如，n 次)，利用

```
glAccum(GL_ACCUM, 1.0/n);
```

累积数据，并最后调用 glAccum (GL_RETURN，1.0);

应该注意，在通过循环的第一遍时，如果利用 GL_LOAD 和省略清除累积缓存，则这个方法较快。利用这个程序在绘最后图像前绘图像 n 次。如果不显示中间图像，则进入不显示颜色缓存中绘图，在此累积，并调用 GL_RETURN 进入显示缓存中绘图(或进入相继与前缓存交换的后缓存中)。

另外，可以给出一个随每个附加部分的累积而显示视图改进的用户接口，并在对图像满意时停止这个过程。具体做法是：在绘相继图像的循环中，每次累积后，调用 glAccum()，设置参数为 GL_RETURN， 设置第二个参数为 16.0/1.0，16.0/2.0，16.0/3.0 等。利用这种方法，在 1 遍后，显示最后图像的 1/16，在 2 遍后，显示 2/16，依次类推。在 GL_RETURN 后，程序应检查是否需要中断这个过程。这个接口是较慢，因为所得到的图像必须在每遍后进行拷贝。

为了确定 n 应是多少，必须折中速度(绘场景次数较多，得到的最后图像所需时间也较多)和质量(绘场景次数较多，得到的图像就较平滑，以利用最大累积缓存的分辨率为限)。

第 14 章　巧用显示列表

OpenGL 显示列表是由一组预先存储起来的留待以后调用的 OpenGL 函数语言组成的，当调用这张显示列表时就依次执行表中所列出的函数语句。给出函数命令，OpenGL 立即执行的绘图方式叫作立即方式。

注意：*显示列表并不同于子函数。显示列表是存储在缓存中的一段固定的命令，它一旦定义好就是不可更改的，因此不能通过更改显示列表中的参数来实现不同的绘制效果，而子函数却只是一段程序代码的有序集合，它可以根据调用参数的变化得到不同的结果。*

OpenGL 显示列表的设计能优化程序运行性能，尤其是网络性能。显示列表实际上是一种高速缓存，而不是动态数据库缓存。也就是说，一旦建立了显示列表，就不能修改它。因为如果要求显示列表可以被修改，则用于显示列表的搜索和内存管理的执行开销会降低性能。

采用显示列表方式绘图一般要比立即方式快，可以大大提高网络性能，即当通过网络发出绘图命令时，由于显示列表驻留在服务器中，因而使网络的负担减轻到最小。另外，在单用户机器上，显示列表同样可以提高效率。因为一旦显示列表被处理成适合图形硬件的格式，则不同的 OpenGL 实现对命令的优化程度将会不同。例如，旋转矩阵函数 glRotate*()，若将它置于显示列表中，则可大大提高性能。因为旋转矩阵有较多的计算量，而在显示列表中，只需存储最后的旋转矩阵，所以甚至能将相邻的几个变换命令并置为单个矩阵相乘。

通过本章的学习，用户将了解显示列表的设计原理，掌握如何去建立和执行显示列表，以及如何去管理显示列表及其索引，本章还对显示列表进行较为深入的讨论，讨论执行多个显示列表，以及用于改变封装方式的显示列表的方法。

14.1　显示列表的基本概念

OpenGL 的显示列表是为了优化性能而设计的，特别在客户/服务器网络环境中，显示列表总是放在服务器上，所以利用显示列表可以减少网络信息流通量。

就优化性能而言，OpenGL 的显示列表是高速缓存中的命令，而不是动态数据库。显示列表一经建立，就不能修改。通常显示列表的内容被预处理，所以执行同一组命令的显示列表的效率比立即执行要高。特别地，通过网络发出 OpenGL 例程时，性能有明显提高。显示列表的预处理对计算密集型的命令尤为重要。

即使在本机上利用显示列表也是有效的，因为建立显示列表可以还将图形绘制命令处理为图形硬件更可兼容的形式。例如，glRotate*()命令，根据特定的 OpenGL 实现，在显示列表中可表现为重的改进。由于旋转矩阵有较多的计算量，而在显示列表中，只需存储最后

的旋转矩阵，所以甚至能将相邻的几个变换命令并置为单个矩阵相乘。利用显示列表有可能在以下几方面获得较大程度的性能优化：

- 矩阵操作。
- 光栅位图和图像。
- 光照、材料性质和光照模型。
- 纹理。
- 多边形点画模式。

当然显示列表也存在某些缺点。一个缺点是执行显示列表比用直接命令方式执行需要的存储要多；另一个缺点是显示列表内容具有不变性。此外，为了优化性能，OpenGL 的显示列表不能被改变，而且显示列表的内容也不能被读出。

为了达到对显示列表和普通的函数调用有一个清楚的区分，下面给出一种用一组语句来绘制 100 个圆的程序：

```
drawCircle()
{
  GLint i;
  GLfloat cosine, sine;

  glBegin(GL_POLYGON);
    for(i=0; i<100; i++){
      cosine=cos(i*2*PI/100.0);
      sine=sin(i*2*PI/100.0);
      glVertex2f(cosine,sine);
    }
  glEnd();
}
```

从程序的执行过程可以看到，每调用一次 drawCircle 函数，程序都会重新计算 cosine 和 sine 变量的值，这大大降低了程序的执行性能。当然，可以先计算这些值，将其存储在数组中，改写成以下程序：

```
drawCircle()
{
  GLint i;
  GLfloat cosine, sine;
  static GLfloat circoords[100][2];
  static GLint inited=0;

  if(inited==0){
    inited=1;
    for(i=0; i<100; i++){
      circcoords[i][0]=cos(i*2*PI/100.0);
```

```
            circcoords[i][1]=sin(i*2*PI/100.0);
        }
    }
    glBegin(GL_POLYGON);
        for(i=0; i<100; i++)
            glVertex2fv(&circcoords[i][0]);
    glEnd();
}
```

这样，虽则程序的性能有了一定的提高，但由于对 for 循环控制变量 i 的递增以及判断是否越界仍然需要耗去一定的计算量，因此程序性能的发挥仍未达到最佳。在使用显示列表时，就只需定义一次如何绘制圆，然后让 OpenGL 记住是如何绘制的，以后每次调用显示列表，OpenGL 只要再按这种方法绘制就行了。下面就给出创建这样一个显示列表的方法：

```
#define MY_CIRCLE_LIST 1

buildCircle()
{
    GLint i;
    GLfloat cosine, sine;

    glNewList(MY_CIRCLE_LIST, GL_COMPILE);
        glBegin(GL_POLYGON);
            for(i=0; i<100; i++){
                cosine=cos(i*2*PI/100.0);
                sine=sin(i*2*PI/100.0);
                glVertex2f(cosine,sine);
            }
        glEnd();
    glEndList();
}
//.............................................
```

显示列表定义好之后，并不能对其进行编辑的，也就是说，它的内容是无法改变的。可以删除和新建一个显示列表，但不能对已有的显示列表进行修改。

注意： 用于定义显示列表的绘制语句应该夹在 glNewList()和 glEndList()两个函数之间，在此之外的 OpenGL 语句并不在显示列表的内容之列，因此调用显示列表时也不会用它们来进行绘图。虽然看上去显示列表的生成过程中也用到了一些运算，但与函数定义完全不同，显示列表只在定义时进行一次运算，当调用它来绘制时，直接调用存储的内容来产生绘制。

14.2 显示列表的建立、执行和管理

显示列表必须在建立之后才能使用，这又是和函数定义不一样的一个地方，函数只要声明和定义之后，在需要使用的地方调用它即可，而显示列表的使用则不然，它不仅需要先定义绘制语句，而且需要在使用之前调用这段建立显示列表的程序。它需要先调用建立显示列表的一段代码，然后在使用时再调用这个显示列表即可。

14.2.1 显示列表的建立

建立和使用 OpenGL 中用于建立显示列表的 2 个函数是：glNewList 和 glEndList。

```
void glNewList(GLuint list, GLenum mode);
```

以上函数用于指定显示列表定义的开始，它带 2 个参数，其中，list 指定显示列表的名称，用于标识显示列表的唯一整数。mode 指定编译方式，它有 2 种取值，

GL_COMPILE　　只编译命令。

GL_COMPILE_AND_EXECUTE　　编译命令建立显示列表后执行命令。

void　glEndList(void)函数用于标记显示列表的结束。

用 glNewList()建立显示列表，在显示列表中按发出的顺序放置相继的命令，直到调用 glEndList()后为止。

在 glNewList()和 glEndList()之间，只能建立一个显示列表。没有 glNewList()而直接调用 glEndList()时，将产生错误：GL_INVALID_OPERATION。

建立显示列表时，只有表达式的值被存放在列表中。因此，即使接着改变数组中的值，也改变显示列表的值。在下面的程序段中，显示列表内含有一个颜色命令，设置当前颜色为黑色(0.0，0.0，0.0)。接着，在显示列表建立之后改变 color_vecter 数组中的值为红色(1.0，0.0，0.0)，由于显示列表实际包含的值是在建立显示列表时的值，所以这种改变是不起作用的。例 14.1 给出一个使用显示列表的简单程序。

例 14.1　使用显示列表

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

GLuint listName = 1;

void myinit (void)
{
glNewList (listName, GL_COMPILE);
       glColor3f(1.0, 0.0, 0.0);
       glBegin (GL_TRIANGLES);
```

```
            glVertex2f (0.0, 0.0);
            glVertex2f (1.0, 0.0);
            glVertex2f (0.0, 1.0);
        glEnd ();
        glTranslatef (1.5, 0.0, 0.0);
    glEndList ();
    glShadeModel (GL_FLAT);
}

void drawLine (void)
{
    glBegin (GL_LINES);
        glVertex2f (0.0, 0.5);
        glVertex2f (15.0, 0.5);
    glEnd ();
}

void display(void)
{
    GLuint i;
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    for (i = 0; i < 10; i++)
        glCallList (listName);
    drawLine ();
    glFlush ();
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D (0.0, 2.0, -0.5 * (GLfloat) h/(GLfloat) w,
            1.5 * (GLfloat) h/(GLfloat) w);
    else
        gluOrtho2D (0.0, 2.0 * (GLfloat) w/(GLfloat) h, -0.5,
            1.5);
```

```
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }

    int main(int argc, char** argv)
    {
        auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
        auxInitPosition (0, 0, 400, 50);
        auxInitWindow (argv[0]);
        myinit ();
        auxReshapeFunc (myReshape);
        auxMainLoop(display);
    }
```

在显示列表程序段中调用了 glTranslatef()函数改变下一个物体将要绘制的位置，如果没有这一条语句，则两次调用显示列表的结果是两次绘制的内容完全重合在一起。在立即方式下调用的 drawLine()函数，同样受到了 10 次 glTranslatef()函数调用的影响。因此，如果在显示列表中调用变换命令，千万别忘了考虑到它们对后续程序的影响。

14.2.2　显示列表中的存储内容

当建立一个显示列表时，列表中只保存表达式的值。因此，如果在调用显示列表进行绘制时，即使数组的内容改变了，绘制的结果也不会发生改变。在下面的程序段中，显示列表包含了设置当前颜色为黑色(0.0, 0.0, 0.0)的语句。在后面的程序中，虽然将 color_vector 的内容改变为红色(1.0, 0.0, 0.0)，但对显示列表的绘制结果并无影响。这应该能够帮助更好地理解显示列表和函数定义的区别。

```
GLfloat color_vector[3]={0.0,0.0,0.0};
glNewList(1,GL_COMPILE);
glColor3fv(color_vector);
glEndList();
color_vector[0]=1.0;
```

并不是所有的 OpenGL 命令都可以存储在显示列表内执行。一般来说，对于通过引用传送参数或带返回值的命令是不能存储在显示列表中的，因为显示列表可能在初始定义的参数的作用域外被调用。当形成显示列表调用这些命令时，它们就被立即执行，而不被存储于显示列表中。下面列出不能存储在显示列表中的 OpenGL 命令：

```
glDeleteLists();
glIsEnabled();
glFeedbackBuffer();
glIsList();
glFinish();
```

```
glPixelStore();
glFlush();
glReadPixels();
glGenLists();
glRenderMode();
glGet*();
glSelectBuffer();
```

注意：如果在建立显示列表时，调用以上这些命令，则 glNewList()将会产生错误。

在网络环境下使用 OpenGL，即客户端和服务器端可以分别位于不同的机器。在创建好一个显示列表之后，它保留在服务器端，因此服务器端不依赖于客户端的任何信息就可以执行显示列表。因此，如果在显示列表中支持诸如 glGet*()和 glIs*()之类的查询命令，主调用的程序会对从网络上得到的因时间而变化的随机性数据而感到奇怪。如果在发送数据之前不对显示列表进行处理，则调用函数就不会知道将数据放在何处。因此，任何产生返回值的 OpenGL 命令都不能存储在显示列表中。其他的函数，诸如 glFlush()和 glFinish()也无法在显示列表中存储，因为它们的执行依赖于客户端的状态。最后，由客户端维护的改变状态信息的命令也不能存储在显示列表中。

14.2.3 执行显示列表

在建立显示列表后，即可通过调用 glCallList()来执行显示列表。当然，同一个显示列表可执行多次，并且执行显示列表和执行立即方式图形可混合调用。

void glCallList(Gluint list)；函数用于执行显示列表，其中 list 指定要执行的显示列表名称(整数)。

由于显示列表含有改变 OpenGL 状态变量值的调用，所以这些值随显示列表的执行而变化，这些命令如同用立即方式执行一样。在完成显示列表的执行后，OpenGL 状态的改变继续存在。但有时需要保存执行显示列表前的状态变量的值，而后在执行显示列表后恢复它们的值 。可以利用 glPushAttrib()保存一组状态变量，而后利用 glPopAttrib()恢复这些值。

例 14.2 中在显示列表中设置了颜色并改变了当前矩阵，那么在调用该显示列表之后，这些影响将依然存在。

例 14.2 执行显示列表后状态改变持续生效

```
glNewList(listIndex,GL_COMPILE);
  glColor3f(1.0, 0.0, 0.0);
  glBegin(GL_POLYGON);
    glVertex2f(0.0,0.0);
    glVertex2f(1.0,0.0);
    glVertex2f(0.0,1.0);
  glEnd();
  glTranslatef(1.5,0.0,0.0);
glEndList();
```

有时候，希望保持状态的改变，而有时候却希望能够保存执行显示列表之前的一些状态，而在执行显示列表之后再恢复这些值。这时可以调用 glPushAttrib()函数来存储一组状态变量，然后再调用 glPopAttrib()函数来恢复它们。要保存和恢复当前矩阵，可以使用 glPushMatrix()函数和 glPopMatrix()函数。例 14.3 中演示了以上函数的用法。

例 14.3 在显示列表中恢复状态变量

```
glNewList(listIndex,GL_COMPILE);
   glPushMatrix();
   glPushAttrib(GL_CURRENT_BIT);
      glColor3f(1.0, 0.0, 0.0);
      glBegin(GL_POLYGON);
         glVertex2f(0.0,0.0);
         glVertex2f(1.0,0.0);
         glVertex2f(0.0,1.0);
      glEnd();
   glTranslatef(1.5,0.0,0.0);
   glPopAttrib();
   glPopMatrix();
glEndList();
```

例 14.4 中绘制了一个绿色的未经平移的直线。如果使用例 14.2 中的不保存和恢复一些状态值的显示列表，则绘制出的直线的颜色将是红色的，显示的位置也将发生变化。

例 14.4 使用显示列表恢复状态变量

```
void display(void)
{
    GLint i;

    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    for (i = 0; i < 10; i++)
      glCallList (listIndex);
    drawLine ();
    glFlush ();
}
```

只要 OpenGL 现场仍然是激活的，就可以从程序内任何地点调用 glCallList()。由于标识显示列表的索引是唯一的，所以在一个子程序中建立的显示列表可在另一个子程序中被执行。显示列表的内容不能保存在数据文件中，也不能用文件来建立显示列表。因而，显示列表是为临时使用的设计。

14.2.4 显示列表的层次

创建一个层次式显示列表，可以在一个显示列表的 glNewList()和 glEndList()函数对之间执行另外一个显示列表。层次显示列表用于由各种构件组成的对象。例如，绘制小汽车的显示列表调用小汽车构件的其他显示列表，绘制车轮的显示列表可以成为绘制小汽车显示列表的子列表：

```
glNewList(listIndex,GL_COMPILE);
  glCallList(handlebars);
  glCallList(frame);
  glTranslatef(1.0,0.0,0.0);
  glCallList(wheel);
  glTranslatef(3.0,0.0,0.0);
  glCallList(wheel);
glEndList();
glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data);
```

为了避免无限递归，对显示列表的嵌套层有一定限制。其限制至少为 64，根据实现的需要，嵌套层可以较多。可以通过调用以下函数确定 OpenGL 实现的嵌套限制：

```
glGetIntegerv (GL_MAX_LIST_NESTING,  GLint * data);
```

OpenGL 允许建立一个显示列表来调用另一个还未建立的显示列表，但当第一个表调用第二个未定义的表后，什么也不发生。

利用一个含有较低层表的层次显示列表，使之成为可编辑的显示列表。例如，利用一个显示列表绘多边形，使多边形的顶点易于编辑。要编辑一个顶点，只需重新建立相应顶点的单个显示列表。用相同索引建立新显示列表自动删除老显示列表。这种方法不一定得到存储器使用性能或峰值性能的优化，但在某些情况下是有用的。当利用任何正整数作为显示列表索引时，有可能偶然选择一个相同的索引，而删除了现有的显示列表。为了防止偶然删除显示列表，利用 glGenLists()生成一个未用过的索引，利用 glIsList()确定某个指定的索引是否正在使用中。此外，还可利用 DeleteLists()明确删除某个指定的显示列表或一批显示列表。例如，如果希望在显示列表中绘制一个多边形，并允许在程序中编辑它的顶点，那么可以参见下面的例 14.5。

例 14.5 使用层次显示列表

```
glNewList(1,GL_COMPILE);
glVertex3f(v1);
glEndList();
glNewList(2,GL_COMPILE);
glVertex3f(v2);
glEndList();
glNewList(3,GL_COMPILE);
glVertex3f(v3);
```

```
glEndList();

glNewList(4,GL_COMPILE);
glBegin(GL_POLYGON);
glCallList(1);
glCallList(2);
glCallList(3);
glEnd();
glEndList();
```

为了绘制这个多边形，调用第 4 号显示列表。若要编辑一个顶点，只要重新创建与那个顶点有关的显示列表即可。由于一个索引值唯一确定一个显示列表，因此创建一个于已有显示列表相同索引值的显示列表，将自动将原来的显示列表删除。当然这种方法不一定能保证对内存的优化使用甚至使之达到峰值效果，但是在一些情况下其结果还是可以接受的。

14.2.5 管理显示列表及其索引

显示列表能支持层次调用，可以通过在 glNewList()和 glEndList()对之间调用 glCallList() 产生另一个显示列表，也就是说，被调用的显示列表成为了主调用显示列表的子列表。生成一批相邻的空显示列表索引。

GLuiut gl GenLists(GLizei range);

range 指定要生成的相邻空显示列表索引的数目。

glGenLists()返回一个整数，使生成的 range 个相邻空显示列表索引为 n ，n+1，n+range.1。当请求的索引数无效，或 range 为 0 时，glGenLists()返回 0。函数 glIsList()用于检验显示列表的存在。其中 list 参数可以指定某个可能的显示列表名。

当 list 已用作某个显示列表索引时，返回 TRUE；否则，返回 FALSE。

下面是请求单个索引的例子，在检证索引为有效之后，可以用这个索引建立一个新的显示列表：

例 14.6 定义多个显示列表

```
void initStrokedFont(void)
{
   GLuint base;

   base = glGenLists (128);
   glListBase(base);
   glNewList(base+'A', GL_COMPILE);
     drawLetter(Adata); glEndList();
   glNewList(base+'E', GL_COMPILE);
     drawLetter(Edata); glEndList();
   glNewList(base+'P', GL_COMPILE);
```

```
    drawLetter(Pdata);  glEndList();
  glNewList(base+'R', GL_COMPILE);
    drawLetter(Rdata);  glEndList();
  glNewList(base+'S', GL_COMPILE);
    drawLetter(Sdata);  glEndList();
  glNewList(base+' ', GL_COMPILE);     /* 空格字符 */
    glTranslatef(8.0, 0.0, 0.0);  glEndList();
}
```

命令 glDeleteLists()删除一组相邻的显示列表，从而使这些索引可再利用。

删除一组相邻的显示列表。

list　指定要删除的第一个显示列表的索引。

range　指定要删除的显示列表的数目。

OpenGL 提供连续执行几个显示列表的高效机制。这种机制需要将显示列表索引置于数组中，调用 glCallLists()。执行多个显示列表的最佳方法是让显示列表索引对应于一批有意义的值。例如，建立一种字体，每个显示列表索引可对应该字体中某个字符的 ASCII 值 。如果需要几个字体，就必须对每个字体建立不同的初始索引。在调用 glCallLists()之前，利用 glListBase()可指定这个初始索引。

对 glCallLists()设置显示列表的基。

base　指定一个整数的偏移值(offset)，将偏移值加到 glCallLists()中，显示列表索引上生成最后的显示列表索引，初始值为 0。

表基对 glCallList()没有作用，glCallList()只执行一个显示列表。

执行 n 个显示列表。

n　指定要执行的显示列表的数目。

type 指定 lists 数组中值的类型和每元素的“跨距”(或大小)。这个参数通常为下列常数中的一个：GL_2_BYTES,GL_3_BYTES 或 GL_4_BYTES，这些情况利用如下算法：位移 2 个、3 个或 4 个字节的序列，逐个字节相加起来计算出显示列表的偏移值。意即对多个字节数据，从数组中按最高位数据在先的顺序取字节。

lists　指定 n 显示列表索引的数组。这个参数指针类型为 void，因为偏移值 可以是字节、短整数、长整数，或浮点数，根据 type 的值而不同。

要执行的显示列表索引计算方法为：把当前显示列表的基所指定的偏移值(用 glListBase()指定)与 lists 数组中显示列表索引相加。

下面说明一个应用多个显示列表的例子。首先列出程序中的一段，这段程序利用笔划式字体(由线段组成的字母集)绘字符。inistStrokedFont()例程对每个字母设置显示列表索引，使这些索引对应相应的 ASCII 值。

程序段定义多个显示列表。

例 14.7　使用多个显示列表定义一组字体

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
```

```
#define PT 1
#define STROKE 2
#define END 3

typedef struct charpoint {
   GLfloat  x, y;
   int   type;
} CP;

CP Adata[] = {
   { 0, 0, PT}, {0, 9, PT}, {1, 10, PT}, {4, 10, PT},
   {5, 9, PT}, {5, 0, STROKE}, {0, 5, PT}, {5, 5, END}
};

CP Edata[] = {
   {5, 0, PT}, {0, 0, PT}, {0, 10, PT}, {5, 10, STROKE},
   {0, 5, PT}, {4, 5, END}
};

CP Pdata[] = {
   {0, 0, PT}, {0, 10, PT},  {4, 10, PT}, {5, 9, PT},
   {5, 6, PT}, {4, 5, PT}, {0, 5, END}
};

CP Rdata[] = {
   {0, 0, PT}, {0, 10, PT},  {4, 10, PT}, {5, 9, PT},
   {5, 6, PT}, {4, 5, PT}, {0, 5, STROKE}, {3, 5, PT},
   {5, 0, END}
};
CP Sdata[] = {
   {0, 1, PT}, {1, 0, PT}, {4, 0, PT}, {5, 1, PT}, {5, 4, PT},
   {4, 5, PT}, {1, 5, PT}, {0, 6, PT}, {0, 9, PT}, {1, 10, PT},
   {4, 10, PT}, {5, 9, END}
};

void drawLetter(CP *l)
{
   glBegin(GL_LINE_STRIP);
```

```
    while (1) {
        switch (l.>type) {
            case PT:
                glVertex2fv(&l->x);
                break;
            case STROKE:
                glVertex2fv(&l->x);
                glEnd();
                glBegin(GL_LINE_STRIP);
                break;
            case END:
                glVertex2fv(&l->x);
                glEnd();
                glTranslatef(8.0, 0.0, 0.0);
            return;
        }
    l++;
    }
}

void myinit (void)
{
    GLuint base;

    glShadeModel (GL_FLAT);

    base = glGenLists (128);
    glListBase(base);
    glNewList(base+'A', GL_COMPILE); drawLetter(Adata);
        glEndList();
    glNewList(base+'E', GL_COMPILE); drawLetter(Edata);
        glEndList();
    glNewList(base+'P', GL_COMPILE); drawLetter(Pdata);
        glEndList();
    glNewList(base+'R', GL_COMPILE); drawLetter(Rdata);
        glEndList();
    glNewList(base+'S', GL_COMPILE); drawLetter(Sdata);
        glEndList();
    glNewList(base+'', GL_COMPILE);
```

```
        glTranslatef(8.0, 0.0, 0.0); glEndList();
}

char *test1 = "A SPARE SERAPE APPEARS AS";
char *test2 = "APES PREPARE RARE PEPPERS";

void printStrokedString(char *s)
{
    GLsizei len = strlen(s);
    glCallLists(len, GL_BYTE, (GLbyte *)s);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0, 30.0, 0.0);
    printStrokedString(test1);
    glPopMatrix();
    glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0, 13.0, 0.0);
    printStrokedString(test2);
    glPopMatrix();
    glFlush();
}


int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 440, 120);
    auxInitWindow (argv[0]);
    myinit ();
    auxMainLoop(display);
}
```

glGenLists()命令分配 128 个相邻显示索引。相邻索引中第一个成为显示列表的基。对

每个字母作成一个显示列表，每个显示列表索引是基和该字母的 ASII 值之和。本例只建立几个字母和空字符。

在建立显示列表后，即可调用 glCallLists()执行显示列表。例如，传送一个字符串给 printStrokedstring()。

利用字符串中每个字母的 ASCII 值作为进入显示列表索引中的偏移值。将当前表基加到每个字母的 ASCII 值上，以确定要执行最后的显示列表索引。

14.3　改变封装方式的显示列表

利用显示列表可以编排和存放一批命令，用以改变各方式，或设置各种参数。当需要从一组设置转换到另一组设置时，利用显示列表比直接调用更为有效，因为这些设置可以用一种符合图形系统要求的格式来存储。

当在几个纹理映射之间转换时，利用显示列表是较有效的(纹理映射在第 9 章中讨论)。假设有两种相当大的不同纹理，在利用纹理时两种纹理都要装入纹理存储器中，不用显示列表时，必须装第一个纹理数据，而后利用它绘图。等待第二个纹理被装入存储器中，而后利用它绘图。当需要切回到一个纹理时，还必须再装入到存储器中，而不是从纹理存储器中提取。因为不用显示列表机制给出“控制符”(handle)来标识它，OpenGL 就无法知道纹理已被存储。利用显示列表，两种纹理能一次装入到纹理存储器中，当需要使用时不必重新装入。

显示列表可能比立即方式较有效的另一种情况是：有关的各种光照、光照模型和材料参数设置之间的转换。显示列表还用于点画模式、雾参数，以及裁剪面方程等。

下面的程序说明如何利用显示列表在三种不同的线型之间转换。首先，调用 glGenLists()对每种线型分配一个显示列表和建立一个显示列表。而后，利用 glGallLists()从一种线型转换到另一种。

例 14.8　使用显示列表进行模式改变

```
GLuint offset;
offset = glGenLists (3);

glNewList (offset, GL_COMPILE);
   glDisable (GL_LINE_STIPPLE);
glEndList ();

glNewList (offset+1, GL_COMPILE);
   glEnable (GL_LINE_STIPPLE);
   glLineStipple (1, 0x0F0F);
glEndList ();

glNewList (offset+2, GL_COMPILE);
   glEnable (GL_LINE_STIPPLE);
```

```
   glLineStipple (1, 0x1111);
glEndList ();

#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
   glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();

glCallList (offset);
drawOneLine (50.0, 125.0, 350.0, 125.0);

glCallList (offset+1);
drawOneLine (50.0, 100.0, 350.0, 100.0);

glCallList (offset+2);
drawOneLine (50.0, 75.0, 350.0, 75.0);
```

第 15 章　参数曲面和曲线编程

本章将介绍以下内容：

- 利用 OpenGL 的鉴别器命令绘制基本的曲线和曲面。
- 利用 GLU 的高层 NURBS 功能绘制更复杂的曲线和曲面。

在最低层，图形硬件可以绘制点、线段以及简单多边形(例如，三角形和四边形)，然后通过使用大量小线段或多边形来近似绘出平滑的曲线和曲面。但在数学上，我们可以利用少量的参数，例如，用 n 个控制点来描绘许多有用的曲线和曲面。保存一个 Bézier 曲面只需 16 个控制点，这比保存 1000 个三角形及每个顶点的法向量信息所需的存储量要少得多，并且，即使使用 1000 个三角形，也只能逼近真实曲面，而控制点则能精确地描绘曲面。

OpenGL 中的鉴别器就提供了一种仅使用控制点来指定曲线或曲面上的点的方法。在指定了控制点之后，可以用任意精度来绘制曲线和曲面。另外，还能对曲面自动计算法向量。有很多方法可以利用鉴别器生成的点=如绘制曲面上的点，绘制线框曲面，或绘制全局光照和明暗处理的曲面。

利用鉴别器可以描绘任何多项式或有理多项式样条或任何次数的曲面，几乎包括目前使用的全部样条和样条曲面，如 B 样条，NURBS(非均匀有理 B 样条)曲面，Bézier 曲线和曲面，以及 Hermite 样条。但由于鉴别器只提供曲线和曲面的低层描述，所以一般在实用库下使用它们，实用库提供了一个较高层的接口。GLU 的 NURBS 设施就是较高层的接口，NURBS 例程封状大量复杂的程序，但最后的绘制是利用鉴别器执行的。

1. **定义鉴别器**

指定：鉴别器类型，每个方向的起止范围、次数(控制点数)，每个方向步进一个单位对应的浮点值个数控制点清单。

2. **打开鉴别器**

viod glEnable(鉴别器类型)。

3. **引用鉴别器**

glEvalCoord(Ui,Vi);当前生成的点的值域注意与 1、中取得一致，如果总的值域为[0, 1]，则 100 个点组成的曲线上第 10 点的值域为(1.0)*10/100。

Bézier 曲线是一个单变量矢量函数：C(u) = [X(u) Y(u) Z(u)]。其中，u 在某个值域中变化(如[0,1])，Bézier 曲面片是双变量矢量函数：S(u,v) = [X(u,v) Y(u,v) Z(u,v)] 其中 u 和 v 都在某个值域中变化。排列不一定是三维的。例如，在一个平面或纹理坐标中输出二维输出值，或在描述 RGBA 信息时得到四维值，在处理灰度级时一维输出值都是合理的。

对每个 u(或 u 和 v，在曲面中)，C()或 S()的公式计算得出曲线或曲面上的一个点。要使用一个鉴别器，首先定义函数 C()或 S()，启用它(Enable)，然后用 glEvalCoord1()或 glEvalCoord2()命令取代 glVertex()。这样，曲线或曲面的顶点就可以和其他任何顶点一样地使用，如用于形成点或线。此外，一些命令自动生成一系列的顶点，用以生成一个在 u 上等分的网格。一维和二维鉴别器是相似的，但一维的描述要简单些。

一维鉴别器：

这部分提供了一个使用一维鉴别器绘制一个曲线的例子。其中控制曲线的的命令和公式稍后说明。

一维例子：一个简单的 Bézier 曲线。

例 15.1 所展示的程序用 4 个控制点绘制一条立方 Bézier 曲线，如图 15.1 所示。

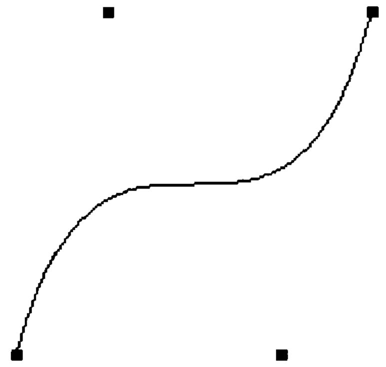

图15.1 一条Bézier曲线

例 15.1 用 4 个控制点绘制一条立方 Bézier 曲线 bezcurve.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
GLfloat ctrlpoints[4][3] = { //定义控制点
{ -4.0, -4.0, 0.0}, { .2.0, 4.0, 0.0},
{2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};
void myinit(void)
{
glClearColor(0.0, 0.0, 0.0, 1.0);
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4,&ctrlpoints[0][0]);
glEnable(GL_MAP1_VERTEX_3);
glShadeModel(GL_FLAT);
}
void display(void)
{
int i;
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_LINE_STRIP);
for (i = 0; i <= 30; i++)
glEvalCoord1f((GLfloat) i/30.0);
glEnd();
// 下面一段代码将控制点显示为黄色的点
glPointSize(5.0);
glColor3f(1.0, 1.0, 0.0);
glBegin(GL_POINTS);
for (i = 0; i < 4; i++)
glVertex3fv(&ctrlpoints[i][0]);
glEnd();
glFlush();
}
void myReshape(GLsizei w, GLsizei h)
{
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
else
glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
int main(int argc, char** argv)
{
auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
auxInitPosition (0, 0, 500, 500);
auxInitWindow (argv[0]);
myinit();
auxReshapeFunc (myReshape);
auxMainLoop(display);
}
```

一条三次 Bézier 曲线由 4 个控制点描述，这些控制点出现在本例的 ctrlpoints[ ][ ]数组中。这个数组是传给 glMap1f()的参数之一。该命令的所有参数如下：

GL_MAP1_VERTEX_3 ：产生三维顶点。

0 ：参数 u 的最小值。

1 ：参数 u 的最大值。

3 ：每前进一个控制点应在生成数据中前进的浮点值个数。

4 ：spline 的次数＋1；在本例中度数是 3(因为是立方曲线)。

&ctrlpoints[0][0] ：指向第一个控制点数据的指针

注意：第二、三个参数控制了曲线的参数化——即变量 u 从 0 变到 1 时曲线从一端走到另一端。对 glEnable()的调用为产生二维顶点而打开(enable)一维鉴别器。

曲线是在例程 display()中的 glBegin()和 glEnd()调用之间绘制的。由于鉴别器是打开的，命令 glEvalCoord1f()就像发出了一条带有坐标参数的 glVertex()命令一样，这些坐标就是曲线上对应于输入参数 u 的顶点的坐标。

定义并评估一个一维鉴别器：

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

n 度(或 n+1 次)伯恩斯坦(Bernstein)多项式由由下式给定：

$$c(u) = \sum_{i=0}^{n} B_i^n(u) p_i$$

如果 Pi 描述一套控制点(1、2、3 维甚至 4 维)，那么，公式：

$$C\left(\frac{u-u_1}{u_2-u_1}\right)$$

在 u 从 0 变到 1 时就描述了一条 Bézier 曲线。要描述相同的曲线而允许 u 在 u1 到 u2 间变化，定值：

命令 glMap1()定义一个使用这些公式的一维鉴别器：

```
void glMap1{fd}(GLenum target, TYPEu1, TYPEu2, GLint stride, Glint order,
                const TYPE *points);
```

定义一个一维鉴别器。Target 参数指明控制点代表什么，清单见表 15.1，以及在 points 中需要提供多少个值。这些点可以代表顶点、RGBA 颜色数据、法向量或纹理坐标。例如，用 GL__MAP1__COLOR__4，鉴别器生成属于某一条曲线的四维色彩空间中的颜色数据。在引用鉴别器之前，用表 15.1 列出的参数值打开每个定义好的鉴别器。传递适当的值给 glEnable()或 glDisable()以打开或禁用鉴别器。

glMap1*()的第 2 个参数，u1 和 u2，指出了变量 u 的范围。变量 stride 是每个存储器块中单精度或双精度值(根据需要)的个数。即，一个控制点开始到下一个控制点开始间的偏移值。

orders 是度数加一，它应当与控制点数一致。pionts 参数指向第一个控制点的第一个坐标。例如，将数据结构传给 glMap1*()时，对 pionts 使用值：(GLfloat)(&ctlpoints[0].x)。

表15.1 glMap1*()函数中所用的控制点类型

参 数	含 义
GL_MAP1_VERTEX_3	x, y, z 顶点坐标
GL_MAP1_VERTEX_4	x, y, z, w 顶点坐标
GL_MAP1_INDEX	颜色索引
GL_MAP1_COLOR_4	R, G, B, A 颜色
GL_MAP1_NORMAL	法向量
GL_MAP1_TEXTURE_COORD_1	s 一维纹理坐标
GL_MAP1_TEXTURE_COORD_2	s, t 二维纹理坐标
GL_MAP1_TEXTURE_COORD_3	s, t, r 三维纹理坐标
GL_MAP1_TEXTURE_COORD_4	s, t, r, q 纹理坐标

一次可以有多个鉴别器被定值。例如，如果同时定义并打开 GL_MAP1_ VERTEX_3 和 GL_MAP1_COLOR_4 两个鉴别器，然后调用 glEvalCoord1()同时生成一个位置和颜色。此时，只可有一个顶点鉴别器打开，尽管或许已经定义 2 个。类似的，只能有一个纹理鉴别器是活动的。然而不同的是，鉴别器可以用于生成(顶点、法向、颜色和纹理坐标)数据的任意组合。如果同一类型的多个鉴别器被定义并打开，只有维数最高的那个被使用。

使用 glEvalCoord1*()以定值一个已经定义并打开的一维图(map)。

```
void glEvalCoord1{fd}{v}(TYPE u);
```

引起对打开的一维图进行定值。参数 u 是坐标值域的值(或指针，用于本命令的向量版)。

定义一维坐标的等分空间值可以用任意的 u 值调用 glEvalCoord1()，但通常使用等分空间值(evenly spaced value)，正如例 15.1 所示。为获取等分空间值，用 glMapGrid1*()定义一个一维格子，用 glEvalMesh1()应用它。

void glMapGrid1{fd}(GLint n, TYPE u1, TYPE u2); 定义一个分 n 步从 u1 进到 u2 的格子，这个格子是等分空间的。

```
void glEvalMesh1(GLenum mode, GLint p1, GLint p2);
```

将当前已经定义的图的格子应用于所有打开的鉴别器。mode 可以是 GL_POINT 或 GL_LINE，这要看看是想画曲线上的点还是相连的线了。这个调用与在[p1，p2]之间每一步发一条 glEvalCoord1()作用完全相同，其中 0<=p1,p2<=n。从程序的角度看，它等价于：

```
glBegin(GL_POINTS); /* OR glBegin(GL_LINE_STRIP); */
for (i = p1; i <= p2; i++)
glEvalCoord1(u1 + i*(u2.u1)/n);
glEnd();
```

例外：如果 i=0 或 i=n，那么 glEvalCoord()将以 u1 或 u2 作为参数被调用。

二维鉴别器

$$s(u,v)=\sum_{i=0}^{n}\sum_{j=0}^{m}B_i^n(u)B_j^m(v)P_{ij}$$

二维的每件事与一维时的情况相似，除了所有的命令都必须用 2 个参数，u 和 v。点、颜色、法向量或纹理坐标都必须作用于一个曲面而非曲线。

使用二维鉴别器的过程与一维相似：

(1) 定义鉴别器，用 glMap2*()。

(2) 打开鉴别器，传适当的值给 glEnable()。

(3) 在 glBegin/glEnd()对之间调用 glEvalCoord2()来引用这些鉴别器。

定义并定值一个二维鉴别器。

用 glMap2*()和 glEvalCoord2*()以定义然后引用一个一维鉴别器。

```
void glMap2{fd}(GLenum target, TYPE u1, TYPE u2, GLint ustride, GLint
        uorder,
TYPE v1, TYPE v2, GLint vstride, GLint vorder, TYPE pionts);
```

target 参数可以取表 15.1 中的任何一个值，用 MAP2 代替 MAP1。如前所述，这些值也与 glEnabled()一起用以打开相应的鉴别器。u 和 v 的最小和最多值由 u1,u2,v1,v2 提供。参数 ustride 和 vstride 说明了一套单(双)精度值(看情况)的个数，这套值允许用户从较大的控制点数组中选择一个子块。例如，如果在表单(form)中出现了如下数据：

```
GLfloat ctlpoints[100][100][3];
```

并且想用人 clpoints[20][30]数组前面的 4×4 子集，可以选择 ustride 为 100*3，且 vstride 为 3。起点 points，应该设为&ctlpoints[20][30][0]，最后，次元参数 uorder 和 vorder 可以不同，允许在一个方向上为三次而另一个方向上为二次的曲面片，例如：

```
void glEvalCoord2{fd}{v}(TYPE u, TYPE v);
```

可以对打开的二维图的定值。参数 u 和 v 是坐标值域的值(或指向值的指针，用于矢量形式的命令)。如果顶点鉴别器中的一个被打开了(GL_MAP2_VERTEX_3 或 GL_MAP2_VERTEX_4)，那么曲面的法向量将被分析计算。法向量是与生成的顶点相关联的，该(通过以 GL_AUTO_NORMAL 调用 glEnalbe())。如果它被关闭了，相应的，打开的法向量图将被用于产生一个法向量。如果没有这样的图存在，那么使用当前法向量。

二维例子：一个 Bézier 曲面。

例 15.2 用鉴别器绘制一个线框的 Bézier 曲面，如图 15.2 所示。此例中，曲面是在每个方向上用 9 条曲线绘成的。每条曲线画成 30 段。要得到完整的程序请将示例 15.1 中的 myReshape()和 main()例子加进来。

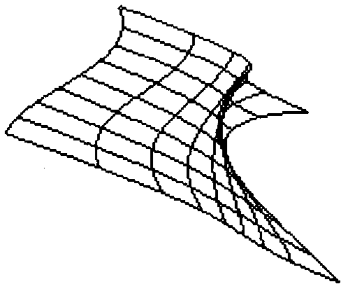

图15.2　一个Bézier曲面

例 15.2　画一个 Bézier 曲面

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
GLfloat ctrlpoints[4][4][3] = {
{{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
{0.5, -1.5, .1.0}, {1.5, -1.5, 2.0}},
{{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
            {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
{{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
{0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
{{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
{0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};
void display(void)
{
int i, j;
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glPushMatrix ();
glRotatef(85.0, 1.0, 1.0, 1.0);
for (j = 0; j <= 8; j++) {
glBegin(GL_LINE_STRIP);
for (i = 0; i <= 30; i++)
glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
glEnd();
glBegin(GL_LINE_STRIP);
for (i = 0; i <= 30; i++)
glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
```

```
glEnd();
}
glPopMatrix ();
glFlush();
}
void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
           0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}
```

下面定义二维空间等分坐标。

在二维中，glMapGrid2*()和 glEvalMesh2()命令与一维版是相似的，但必须同时包含 u 和 v 的信息。

```
void glMapGrid2{fd}(GLint nu, TYPE u1, TYPE u2, GLint nv, TYPE v1,
            TYPE v2);
void glEvalMesh2(GLenum mode, GLint p1, GLint p2, GLint q1, GLint
            q2);
```

定义一个二维图网格，按 nu 个空间等分步从 u1 到 u2、分 nv 步从 v1 到 v2(glMapGrid2*())，然后将这个网格作用于所有打开的鉴别器(glEvalMesh2())。这 2 个命令与一维时唯一明显不同的是，在 glEvalMesh2()中，mode 参数可以是 GL_POINT，GL_LINE 之外的 GL_FILL，它产生用 quad.mesh 图元填充的多边形。确切的说，glEvalMesh2()与下面的代码片断基本等价(基本等价，因为当 i 等于从 nu 或 j 到 nv，那么参数确切地等于 u2 或 v2，而不等于 u1+nu*(u2-u1)/nu，这点可能引起舍入错误)。

```
glBegin(GL_POINTS);  // mode == GL_POINT
for (i = nu1; i <= nu2; i++)
for (j = nv1; j <= nv2; j++)
glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();
```

或者，

```
for (i = nu1; i <= nu2; i++) {
glBegin(GL_LINES);
for (j = nv1; j <= nv2; j++)
glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();
}
```

```
for (j = nv1; j <= nv2; j++)
glBegin(GL_LINES);
for (i = nu1; i <= nu2; i++)
glEvalCoord2(u1 + i*(u2.u1)/nu, v1+j*(v2.v1)/nv);
glEnd();
}
```

或者，

```
for (i = nu1; i < nu2; i++)
glBegin(GL_QUAD_STRIP);
for (j = nv1; j <= nv2; j++)
glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEvalCoord2(u1 + (i+1)*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();
}
```

例 15.3 展示了绘制一个 Bézier 曲面时与例 15.2 的不同之处，仅仅是使用 glMapGrid2() 和 glEvalMesh2()分割值域平面为一个统一的 8×8 网格。这个程序也加入了灯光和阴影，如图 15.3 所示。

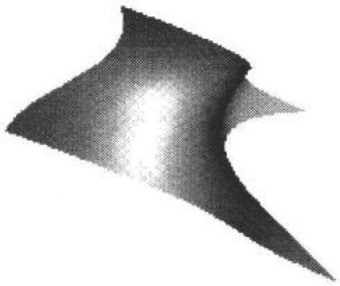

图15.3 利用网格绘制一个加光照和明暗处理的Bézier曲面

例 15.3 绘制一个加光照、明暗处理的 Bézier 曲面(用网格)：bezmesh.c

```
void initlights(void)
{
GLfloat ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat position[] = { 0.0, 0.0, 2.0, 1.0 };
GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[] = { 50.0 };

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

```
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
glLightfv(GL_LIGHT0, GL_POSITION, position);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT_AND_BACK,GL_SHININESS, mat_shininess);
}

void display(void)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glRotatef(85.0, 1.0, 1.0, 1.0);
glEvalMesh2(GL_FILL, 0, 8, 0, 8);
glPopMatrix();
glFlush();
}

void myinit(void)
{
glClearColor (0.0, 0.0, 0.0, 1.0);
glEnable(GL_DEPTH_TEST);
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
0, 1, 12, 4, &ctrlpoints[0][0][0]);
glEnable(GL_MAP2_VERTEX_3);
glEnable(GL_AUTO_NORMAL);
glMapGrid2f(8, 0.0, 1.0, 8, 0.0, 1.0);
initlights();
}
```

第 16 章　选择与反馈实现图形交互

在介绍了前面的许多绘制方法之后，本章将讲解能编写可以实现交互式操作的图形绘制、编辑环境等方面内容。具体如下：

- 建立允许用户选择(select)屏幕区域或拾取(pick)绘制在屏幕上的物体的应用程序。
- 利用 OpenGL 的反馈(feedback)模式获取渲染计算结果。

不同的图形应用程序有不同的应用需求，有些图形应用只需要绘制二维和三维物体的静止图像，而很多应用则要求允许在屏幕上标识对象，然后对其进行移动、修改、删除，或进行其他一些操作。OpenGL 完全提供了对这类交互式应用的支持。由于在屏幕上所绘的对象通常要经过多次旋转、平移和透视变换。而经过多次变换后，需要进行选择操作时，如果不借助特别的机制，就往往难以确定三维场景中哪个对象正在被选中。为了解决这个问题，OpenGL 提供一种选择机制，它能自动得出窗口的指定区域内绘制的是哪些对象。利用这种机制及一个专门的实用程序，可以确定这个区域正在用鼠标指定或拾取的是哪些对象。

同选择机制一样，反馈是另外一种 OpenGL 操作模式。在反馈模式下，可以利用图形硬件和 OpenGL 执行一般的绘制计算。但在反馈模式下，OpenGL 并不利用计算结果直接在屏幕上绘制图像，而是返回(或反馈)计算出的绘图信息。例如，当需要在绘图仪上，而不是屏幕上绘制三维对象时，就需要在反馈模式下绘制景物，这样就可以收集绘图指令，计算出绘制结果，再将其转换为绘图仪能理解的命令输出。

在选择和反馈两种模式下，绘图信息将返回给应用程序，而不是像在绘制模式下那样发送到帧缓存，因此当 OpenGL 处于选择或反馈模式下时，屏幕将保持冻结(即不进行图形绘制)。

本章各节将分别说明这些模式，主要讨论如何利用选择模式和有关例程在屏幕上拾取绘出的对象。另外还将描述如何利用反馈模式获取屏幕上绘图的信息，以及如何将这些信息有效地组织起来。

16.1　选择模式

一般来说，当利用 OpenGL 的选择机制时，首先在缓存中绘制场景，然后进入选择模式并重新绘制场景。但一旦处于选择模式时，帧缓存的内容要等到退出选择模式后才能改变。当退出选择模式时，OpenGL 返回一个与视图体相交的图元表(视图是由当前的 modelview 和投影矩阵及任何指定的裁剪面所定义的)。与视图体相交的每个图元会发生选择命中(hit)。图元表实际返回一个数组，数组中包含了整数值名称和对应于名栈(name stack)当前内容的相关数据(称为命中记录)。当在选择模式下发出了绘图元命令时，将名称装到栈上构成名

栈。当返回名表时，就能利用名表来确定在屏幕上所选择的图元。

除了这个选择机制外，OpenGL 还提供了一个实用程序，用以使选择简化，它可以在某些情况下把绘图限制在视口的小区域内从而对选择进行简化。通常利用这个程序来确定在光标附近绘制哪些对象，使之能识别拾取了哪个对象。此外，还能指定附加的裁剪面确定选择区的边界。

16.1.1 选择的基本步骤

利用选择机制，须执行如下步骤：

(1) 利用 glSelectBuffer()指定存放返回命中记录的数组。

(2) 利用 glRenderMode()将参数设置为 GL_SELECT，进入选择模式。

(3) 利用 glInitNames()和 glPushName()初始化名栈。

(4) 定义选择所用的视图体，通常与初始绘场景所用的视图体是不同的，或许需要利用 glPushMatrix()和 glPopMatrix()来保存和恢复当前的变换状态。

(5) 交替发出绘图元命令和操作名栈的命令，使每个关心的图元有适当的指定名称。

(6) 退出选择模式并处理返回的选择数据(命令记录)。

下面描述 glSelectBuffer()和 glRenderMode()。

```
void glSelectBuffer(GLsizei size , GLuint*buffer);
```

建立选择模式的值缓存。

size　指定 buffer 的大小。

buffer 指定存放返回选择数据的数组。

在进入选择模式前，必须调用 glSelectBuffer()。

```
Lint glRenderMode(GLenum Mode);
```

控制应用处在绘制、选择或反馈模式。

mode　设置模式，有以下 3 种定义值：

GL_RENDER　默认值。正常模式。

GL_SELECT　选择模式。

GL_FEEDBACK　反馈模式。

应用保持一种给定的模式直到调用 glRenderMode()和设置不同参数后为止。在进入选择或反馈模式之前，必须调用 glSelectBuffer()和 glFeedbackBuffer()来指定选择数组和反馈数组。

glRenderMode()的返回值是由调用 glRenderMode()时的绘制模式(不是由 mode 参数)确定的。三种绘制模式的返回值如下：

- GL_RENDER　0。
- GL_SELECT　传送至选择缓存的命中记录数目。
- GL_FEEDBACK　传送至反馈缓存的值(非顶点)的数目。

返回负值意味着选择或反馈数组产生上溢。利用 glGetIntegerv()设置参数 GL_RENDER_MODE，可获取当前的模式。

16.1.2 建立名字栈

如前所述，名字栈是构成返回选择信息的基础。建立名字栈，首先要利用 glInitNames()来初始化名字栈，也就是说，先清除原先的名字栈，当发出相应的绘图命令时，将整数名加到名字栈上。操作栈的命令有：将名称推进栈(glPushName())，将名称弹出栈(glPopName)，以及以不同的名称替换栈顶上的名称(glLoadName())。

名称和对象并不是一一对应的，在本例中，给前面 2 个对象各自一个名称，而第三和第四两个对象共用同一个名称。在这种设置下，当使第 3 和第 4 对象中某个或两者同时都发生选择命中时，只返回一个命中记录。在交互应用中当对几个对象命中记录的处理不必区分时，就可以让这多个对象共用同一个名称。

```
void glInitNames(void);
```

初始化名字栈，即清空名字栈。

```
void glPushName (GLuint name);
void glPopName (void);
```

将 name 推进和弹出名字栈。当推进的名称超过名字栈的容量时，产生错误 GL_STACK_UNDERFLOW；弹出名字栈产生错误 GL_STACK_UNDERFLOW。名字栈深度的容量在不同的系统中可能有所不同，但不管在什么系统中都至少包含 64 个名称。调用 glGetIntegerv()函数，将参数设置为 GL_NAME_STACK_DEPTH，可获取当前系统的名字栈深度。

```
void glLoadName (Gluint name)
```

将 name 装在名字栈上。

name 指定替换名字栈顶值的名称。

当调用 glInitNames()后名字栈清空，如果此时立即调用 glLoadName()，则会产生 GL_INVALID_OPERATION 错误。为了防止发生这种错误，当名字栈初始清空时，紧跟着调用一次 glPushName()以使名字栈不空。

上述 3 个操作名字栈的函数只在选择模式下有效，当不处于选择模式时，glPushName()，glPopName()以及 glLoadName()不产生任何作用。因此，可以在选择和正常绘图 2 种模式下使用同样的绘图程序，而不必为 2 种模式单独编写程序，这样程序就得到大大的简化。

16.1.3 命中记录

在选择模式下，与视图体相交的图元将发生一次选择命中。只要执行名字栈操作命令，或者从调用 glRenderMode()函数后最近时间以来选中了一个记录，则 OpenGL 将命中记录写入到选择数组中。利用这个过程，共用同一名称的对象(例如，由多个图元组成的对象)，不产生多次命中记录。另外，命中记录要等到调用 glRenderMode()后才真正写到数组中。值得注意的是：除几何图元外，由 glRasterPos()产生的有效坐标也能发生一次选择命中。在绘制多边形的情况下，当多边形被剔除时，不发生命中。

每个命中记录由 4 项组成，依次为：

- 当发生命中时，在名字栈上名称的数目。
- 从最近记录命中以来，与视图体相交的图元全部顶点的最小和最大窗口坐标 z 值，这 2 个值位在[0，1]范围内，每个乘以 2^{32}-1，并舍入到最接近的无符号整数。
- 命中时名字栈的内容，最底部的元素最先推入堆栈。

当进入选择模式后，OpenGL 先对选择数组开始的指针进行初始化。以后每发生一次命中则把一个命中记录写到数组中，相应地更新指针。如果写命中记录使数组中值的数目大于调用 glSelectBuffer()时所指定的 size 参数，则 OpenGL 把能够容纳的值记录在数组中，并设置一个上溢标志。当调用 glRenderMode()退出选择模式后，这个命令返回所写的命中记录的数目，清除名字栈，重置上溢标志和栈指针。已设置上溢标志时，返回值为 1。

16.1.4 选择的例子

16.3 节中将给出一个使用选择的例子，该例子绘制 4 个三角形(1 个绿色、1 个红色、和 2 个黄色三角形)并处理相应的命中记录。第 1 个三角形发生命中，第 2 个三角形不发生命中，第 3 个和第 4 个三角形同时发生命中。程序中定义绘制三角形例程(drawTriangle ())和绘制表示视图体的线框盒的例程(draoViewVolume ())。processHits()例程输出选择数组。最后，selectObjects()绘制出选择模式下发生命中记录的三角形。

16.1.5 拾取

拾取(picking)是选择处理的一种扩充，利用选择模式可确定是否拾取对象。拾取对象利用一个特殊的拾取矩阵和投影矩阵，通常将绘图限制在光标附近的视口的小区域中。然后允许某种形式的输入，例如，单击鼠标，初始化选择模式。利用建议的选择模式及特殊的拾取矩阵，使在光标附近所绘的对象发生选择命中。因此，当拾取时，通常确定在光标附近绘制哪些对象。

设置拾取几乎与常规选择模式一样，主要有以下一些不同：

拾取通常由输入设备触发。在下面的一些程序例子中，单击鼠标器调用一个执行拾取的函数。

利用实用例程 gluPickMatrix()对当前矩阵乘以一个特殊的投影矩阵。这个例程应在向矩阵栈乘以投影矩阵之前调用。

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble
            height, GLint viewport[4] );
```

定义一个拾取区。即建立一个限制在视口小区域中绘图的投影矩阵，并用当前矩阵栈乘以这个矩阵。

x，y　　指定窗口坐标中拾取区的中心(通常为光标位置)。

width，height　　依次指定窗口坐标中拾取区的宽度和高度。可认为是拾取设备的灵敏度(sensitivity)。

viewport　　指定当前视口。可调用下面的函数得到：

```
glGetIntegerv  (GL_VIEWPORT, GLint *viewport);
```

gluPickMatrix()所建立的矩阵的实际结果是把裁剪区变换到单位立方体-1<(x，y，z)<1(或-w< (wx， wy， wz) < w)中。拾取矩阵有效执行一个正交变换，把这个单位立方体中的子区映射到单位立方体。由于这个变换是任意的，所以可对不同类的区域进行拾取操作。例如，对窗口中旋转的矩形部分。在有些情况下，可以较容易地通过指定附加裁剪面来定义拾取区。

下面的程序说明简单拾取。这个例子表明如何用多个名称来标识图元的不同部分，在本例中，用行和列选择对象。绘一个 3×3 矩形网格，每个矩形有不同的颜色。board[3][3]数组中每个矩形保持当前蓝色量。单击鼠标器，调用 picksquares()例程识别鼠标器拾取那些矩形。2 个名称由以标识网格中每个矩形，一个是行和另一个是列。然后，单击鼠标器，改变光标位置下的全部正方形的颜色。

16.1.6　利用选择编程的提示

大多数允许交互式编辑几何体的程序都能提供一种利用拾取或项目(items)的编辑机制。对于二维绘图程序(例如，正文编辑程序、页面设计程序、线路设计程序)，利用用户自己的拾取计算来取代利用 OpenGL 的拾取机制可能是较容易的。通常，对二维对象容易找到界限线路图中，OpenGL 样式的拾取可能是相当慢的，但当矩形与屏幕对齐时，利用简单的界限框信息，在这种程序中就能极快地进行拾取。相应的，这种程序写起来也简单得多。

另一个例子，因为只有几何对象发生命中，所以要拾取正文就需要用户建立自己的方法。设置当前的光栅位置是一种几何操作，但仅在当前光栅位置的单个可拾取点上才能有效建立，这个点一般在正文的左下角上。如果编辑程序需要在正文串中操作个别的字符，则可利用某种其他的拾取机制。在拾取模式中，可以在每个字符周围绘制一个小矩形，这种作为特殊字符处理正文的方法显然是较容易的。

假如，确定利用 OpenGL 的拾取，则编制程序及其数据结构，会使在选择或正常绘图模式下易于绘一系列合理对象。当拾取对象时，对所用拾取操作可利用同样的数据结构在屏幕上显示选中内容。另外，考虑是否需要选择多个对象。一种选择多个对象的方法是对每个项存储一位，指出是否选择，但利用这种方法，必需遍历整个项表来查找所选择的项。利用保持一选择项的指针来查找项，以便加速这种搜索。对每个项保留选择位也是一种好的方法。由于在绘整个图形时，可能需要绘不同的选择项(例如不同的颜色，或利用在项周围的选择框)。最后，考虑选择用户接口。允许用户执行如下一些功能：

- 选择一项。
- 扫描选择项组(在下面描述这种特性)。
- 加一项到选择中。
- 加扫描选择到当前选择中。
- 由选择删除一项。
- 从一组重叠项中选择单项。

有关二维绘图程序的一般解决方法按如下执行：

(1) 单击鼠标。

(2) 在一个选择项上单击，并选全部其他的当前选择项。如果光标位在多个项上时，则选择最小的。(在三维中，有许多明确选择的其他策略。)

(3) 单击时没有选中内容，按住按钮拖动光标，然后释放按钮选择屏幕对齐的矩形中的全部选中内容，当按钮按下和放开时在光标位置确定矩形的角。这个过程称为扫描选择(sweep selection)。不在扫描区域中的全部项不被选中。(仅当项完全在扫描区域内，或当项的任何部分落在区域内时，决定是否选择项。完全在内的策略通常执行得最好。)

(4) 按住移位(shift)键，并在不是当前选择的项上单击时，即把该项加到选择表中。在选择的项上单击时，即从选择表中删除该项。

(5) 按下移位键执行扫描选择时，则把扫过的项加到当前选择中。

(6) 在一个极端群集的区域中，通常难以进行扫描选择。按下按钮时，光标可能位于某个项上，通常选择该项。用户可使任何区域为扫描选择。但一般用户接口把按住某项加上鼠标器移动解释为选择加拖动。为了解决这个问题，譬如按住 Alt 键使强制扫描选择。依次，下面一组操作构成扫描选择：按住 Alt 按钮、扫描、放开按钮。当按住按钮时在光标下的项被省略。

(7) 扫描选择时按住移位键，即把包围在扫描区域中的项加到当前选择中。

(8) 在多个项上单击，选择其中合适的项。如果未移动光标(或可能移动不超过一个像素)，而在同一位置上再单击时，则去选原来选择的的项，并选择一个在光标下的不同项。利用在同一点上重复单击，以便循环通过全部可能的项。

不同的规则适用于不同情况。在正文编辑程序中，不必担心有关字符的重叠，而多个字符的选择总是文档中的邻接字符。因此，只需要标记第一个和最后一个选择字符来标识整个选择。对于正文，通常控制选择的最好方法是标识字符间的位置而不是字符本身。当选择的开始和结束是在同一字符对之间时，允许有一个空选择，并且置光标于文档中第一个字符之前，或置于非特殊字符号码和最后字符之后。

在三维编辑程序中，可以给出在选择间旋转和缩放的方法，但对通过可能的选择进行循环的如此复杂设计可能是不必要的。从另一方面来说，在三维中的选择是困难的，因为在屏幕上的光标位置通常不给出其深度的指示。

16.2　反馈模式

反馈模式类似于选择模式，不产生像素和冻结屏幕，即不发生绘图，而是把已绘制图元的信息发回给用户。选择和反馈模式之间的主要区别是送回何种信息。在选择模式下，将指定的名称返回到一个整数值数组。而在反馈模式下，把变换后的图元信息送回到一个浮点值数组中。这些值送回到由权标(tokens)组成的反馈数组中。这些权标指定已被处理和变换的图元类型(点、线、多边形、图像或位图)，以及图元的顶点、颜色或其他数据。返回的值是完全经过光照和视图操作变换后的值。通过调用 glRendrMode()，设置参数为 GL_FEEDBACK，初始化反馈模式。

这里说明如何进入和退出反馈模式：

(1) 调用 glFeedbackBuffer()指定保存反馈信息的数组。这个命令的参数描述何种数据类型及有多少数据写入数组。

(2) 调用 glRenderMode()，设置参数为 GL_FEEDBACK，进入反馈模式。(可省略由 glRenderMode()返回的值。)从这里起到退出反馈模式，不光栅化图元产生像素及不改变帧缓存的内容。

(3) 绘图元。当发出绘图命令时，可以几次调用 glPassThrough()，把权标插入到返回的反馈数组中，借此较容易分析数据。

(4) 当需要返回到正常的绘图模式时，调用 glRdnderMode()，设置参数为 GL_RENDER 退出反馈模式。由 glRenderMode()返回的整数是存储在反馈数组的数目。

(5) 分析反馈数据的缓存。

```
void glFeedbackBuffer (Glsizei size, GLenum type, GLfloat*buffer);
```

建立一个反馈数据的缓存。

size　指定能存储在 buffer 中值的最大数目。

type　指定符号常数，描述 buffer 中每个顶点返回的信息。可能接受的符号常数及其含义在表 16.1 中列出。

表16.1 glFeedbackBuffer中type参数的值

类型参数	坐 标	颜 色	纹 理	总 值
GL_2D	x, y	-	-	2
GL_3D	x, y, z	-	-	3
GL_3D	x, y, z	k	-	3+k
GL_3D_COLOR	x, y, z	k	4	7+k
GL_4D_COLOR_TEXTURE	x, y, z, w	k	4	8+k

buffer 存储数据的数组指针。

在进入反馈模式之前必须调用 glFeedbackBuffer()。在表 16.1 中，在颜色索引模式下 k 为 1，在 RGBA 模式下 k 为 4。

16.2.1 反馈数组

在反馈模式下，每个被光栅化的图元(或当光栅位置有效时)，每次调用 glDrawpixels() 或 glCopyPixels()生成值的信息组被拷贝到反馈数组中。值的数目是由 glFeebdackBuffer()中 type 参数来确定的，如在表 16.1 中列出。利用所绘图元类型的适当值，对无光照的二维或三维图元，其值为 GL_2D 或 GL_3D，对光照的三维图元，其值为 GL_3D_COLOR_TEXTURE 或 GL_4D_COLOR_TEXTURE。

每个反馈值的信息组开始指出图元类型的符号常数，后面是描述图元顶点和有关数据的值。各项也可用像素矩形来写。此外，可以在数组中返回明确生成的通过标记(pass.through makers)，有关标记将在下节中详细说明。表 16.2 说明反馈数组的语法，表 16.1 中已描述

与每个返回顶点有关的数据。一个多边形可返回 n 个顶点。并且，反馈所返回的 x，y，z 坐标为窗口坐标。如果返回 w，则位在裁剪坐标中。对位图和像素矩形，所返回的坐标是当前光栅位置的坐标。注意在表 16.2 中，仅当对线段重置线型时，返回 GL_LINE_RESE_TOKEN。

表16.2　反馈数组的语法

图元类型	符号常数	有关数据
点	GL_POINT_TOKEN	顶点
线	GL_LINE_TOKEN GL_LINE_RESET_TOKEN	顶点 顶点
多边形	GL_POLYGON_TOKEN	n 顶点 顶点…顶点
位图	GL_BITMAP_TOKEN	顶点
像素矩形	GL_DRAW_PIXEL_TOKEN GL_COPY_PIXEL_TOKEN	顶点
通过	GL_PASS_THROUGH_TOKEN	浮点数

16.2.2　在反馈模式下利用标记

在多边形的变换、光照、删除多边形和利用 glPolygonMode()说明多边形之后发生反馈。反馈也可能在多于 3 边的多边形被剖分为三角形之后发生(假如特定的 OpenGL 实现是经过执行这种分解来绘制多边形)。因此，有可能难以识别接受的反馈数据中所绘的图元。为了帮助用户分析反馈数据，调用 glPassThrough()在绘图命令序中根据需要插入标记。例如，利用这些标记可分开从不同图元返回的反馈值。这个命令把 GL_PASS_THROUGH_TOKEN 写入到反馈数组中，后面有作为参数传送的浮点值。

```
void glPassThrough(GLfloat token);
```

在写到反馈数组的值流中插入标记。

token　指定标记浮点值。标记由符号常数 GL_PASS_THROUGH_TOKEN 和 token 组成。

这个命令必须在反馈模式下调用。在 glBegln()和 glEnd()之间调用 glPassThrough()时，产生 GL_INVALID_OPERATION 错误。

16.2.3　反馈的例子

下面的程序例子说明反馈模式的应用。这个程序是在正常绘制模式下绘一个光照的三维场景。然后进入反馈模式，并重绘场景。这个程序绘光照的、无纹理的三维对象，因而反馈数据的类型为 GL_3D_COLOR。由于利用 RGBA 模式，所以每个未裁剪的顶点在反馈缓存中产生 7 个值：x，y，z，r，g，b 和 a。

在反馈模式下，这个程序绘出作为折线部分的 2 条线，然后插入一个通过标记，接着

在(-100.0，-100.0，-100.0)上绘一个点，这个点落在正交视图体之外，因而不将任何值置于反馈数组中。最后，插入另一个通过权标，并绘出另一个点。

例 16.2　创建名字栈

```
// 创建一个名字栈
glInitNames();
glPushName(-1);

glPushMatrix(); /* 保留当前变换状态 */

/*按要求创建高图体 */

glLoadName(1);
drawSomeObject();
glLoadName(2);
drawAnotherObject();
glLoadName(3);
drawYetAnotherObject();
drawJustOneMoreObject();
glPopMatrix ();     /* 恢复以前的变换状态*/
```

第一个图元以 GL_LINE_RESET_TOKEN 开始，指出图元是线段，并且线型是重置的。第二个图元以 GL_LINE_TOKEN 开始，也是一个线段，但线型不重置，因而从前面的线段延伸后保持不变，2 条线各有 2 个顶点，每个顶点在反馈数组中产生 7 个值。应该注意，两条线上全部 4 个顶点的 RGBA 值为(0.84，0.84，0.84，1.0)，这是有最大 A 值的明亮灰色。这些颜色值是曲面中量和光照参数相互作用的结果。

在第一和第二个通过标记之间未生成反馈数据，因而可以推断在 2 个 glPassThrough() 调用之间所绘的任何图元在视图体外被裁剪掉。最后，把绘的点(50.0，50.0，0.0)及其有关数据拷贝到反馈数组中。

在这个例子中，前 2 个被绘制的物体有自己的名字，第 3 和第 4 个共用一个名字。这样，如果第 3 或第 4 个物体中的一个或全部引起一个选择命中，只有一个命中记录返回给你。如果处理命中记录时不想区分各个物体，可以让多个物体共享一个名字。

```
void glInitNames(void);
```

清空名字栈。

```
void glPushName(GLuint name);
```

将 name 压入名字栈。压入名字超过栈容量时将生成一个 GL_STACK_OVERFLOW 错误。名字栈深度因 OpenGL 实现(implementations)的不同而不同，但最少要能容纳 64 个名字。你可以用参数 GL_NAME_STACK_DEPTH 调用 glGetIntegerv()以获取名字栈深度：

```
void glPopName(void);
```

弹出名字栈栈顶的那一个名字。从空栈中弹出名字引发 GL_STACK_UNDERFLOW 错误：

```
void glLoadName(GLuint name);
```

用 name 取代名字栈栈顶的那个名字。如果栈是空的，刚调用过 glInitName()后就是这样，glLoadName()生成一个 GL_INVALID_OPRATION 错误。为避免这种情况，如果栈初始时是空的，那么在调用 glLoadName()之前至少调用一次 glPushName()，以在名字栈中放上点东西。

如果不是在选择模式下，对 glPushName()，glPopName()，glLoadName()的调用将被忽略。这使得在选择模式和正常的渲染模式下用相同的绘制代码大为简化。

16.2.4　命中记录

在选择模式下，被视见体裁剪的每个图元引起一个选择命中。当前一个名字栈控制命令被执行或 glRenderMode()被调用后，OpenGL 将一个命中记录写进选择数组，如果从上一次名字栈操纵或 glRenderMode()调用以来有了一个命中记录的话。这个过程中，共用同样名字的物体，例如，由多个图元组成的物体，不生成多个命中记录。当然，命中记录不保证会被写进数组中直到 glRenderMode()被调用。

除图元之外，glRasterPos()产生的有效坐标也可以引起选择命中。在多边形的情况下，如果它已经被消隐掉的话不会有命中记录出现。

每个命中记录由 4 项组成，依次是：

- 当命中出现时名字栈中的名字数
- 至上次记录的命中以来，被视见体裁剪后的图元的所有顶点的窗口 Z 坐标的最大和最小值本次命中时名字栈的内容，最底元素最前。

当前进入选择模式时，OpenGL 初始化一个指针指向选择数组的起点。每写入一个命中记录，指针相应更新。当写入一个命中记录会使数组中值的个数超过 glSelectBuffer()的 size 参数时，OpenGL 会写入尽可能多的记录并设置一个溢出标志。当用 glRenderMode()退出选择模式时，这条命令返回被写入的记录的个数(包括一条部分记录，如果有)，清除名字栈，复位溢出标识，重置栈指针。如设定溢了出标识，则返回值是-1。

16.3　一个选择的例子

在例 16.2 中，4 个三角形(绿、红、两个黄)在选择模式下绘制，相应的命中记录被处理。第 1 个三角形生成一个命中，第 2 个不生成命中，第 3 第 4 个共同生成一个命中。定义了 2 个例程：绘制三角形(drawTriangle())，绘制一个表示视见体的线框的盒子。processHits()例程打印出选择数组。最后，selectObjects()在选择模式下绘制三角形以生成命中记录。

例 16.2　选择实例： select.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"
void drawTriangle (GLfloat x1, GLfloat y1, GLfloat x2,
```

```
GLfloat y2, GLfloat x3, GLfloat y3, GLfloat z)
{
glBegin (GL_TRIANGLES);
glVertex3f (x1, y1, z);
glVertex3f (x2, y2, z);
glVertex3f (x3, y3, z);
glEnd ();
}

void drawViewVolume (GLfloat x1, GLfloat x2, GLfloat y1,
GLfloat y2, GLfloat z1, GLfloat z2)
{
glColor3f (1.0, 1.0, 1.0);
glBegin (GL_LINE_LOOP);
glVertex3f (x1, y1, -z1);
glVertex3f (x2, y1, -z1);
glVertex3f (x2, y2, -z1);
glVertex3f (x1, y2, -z1);
glEnd ();

glBegin (GL_LINE_LOOP);
glVertex3f (x1, y1, -z2);
glVertex3f (x2, y1, -z2);
glVertex3f (x2, y2, -z2);
glVertex3f (x1, y2, -z2);
glEnd ();

glBegin (GL_LINES); /* 4 条线 */
glVertex3f (x1, y1, -z1);
glVertex3f (x1, y1, -z2);
glVertex3f (x1, y2, -z1);
glVertex3f (x1, y2, -z2);
glVertex3f (x2, y1, -z1);
glVertex3f (x2, y1, -z2);
glVertex3f (x2, y2, -z1);
glVertex3f (x2, y2, -z2);
glEnd ();
}
```

```
void drawScene (void)
{
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective (40.0, 4.0/3.0, 0.01, 100.0);

glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
gluLookAt (7.5, 7.5, 12.5, 2.5, 2.5, -5.0, 0.0, 1.0, 0.0);
glColor3f (0.0, 1.0, 0.0); /* 绿色三角形 */
drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
glColor3f (1.0, 0.0, 0.0); /* 红色三角形*/
drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
glColor3f (1.0, 1.0, 0.0); /* 黄色三角形 */
drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
drawViewVolume (0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
}

void processHits (GLint hits, GLuint buffer[])
{
unsigned int i, j;
GLuint names, *ptr;

printf ("hits = %d\n", hits);
ptr = (GLuint *) buffer;
for (i = 0; i < hits; i++) { /* 对于每一次命中 */
names = *ptr;
printf(" number of names for hit = %d\n", names); ptr++;
printf (" z1 is %u;", *ptr); ptr++;
printf (" z2 is %u\n", *ptr); ptr++;
printf (" the name is ");
for (j = 0; j < names; j++) { /* 对于每一个名称 */
printf ("%d ", *ptr); ptr++;
}
printf ("\n");
}
}
```

```
#define BUFSIZE 512

void selectObjects(void)
{
GLuint selectBuf[BUFSIZE];
GLint hits, viewport[4];

glSelectBuffer (BUFSIZE, selectBuf);
(void) glRenderMode (GL_SELECT);

glInitNames();
glPushName(.1);

glPushMatrix ();
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glLoadName(1);
drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
glLoadName(2);
drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
glLoadName(3);
drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
glPopMatrix ();
glFlush ();

hits = glRenderMode (GL_RENDER);
processHits (hits, selectBuf);
}

void myinit (void)
{
glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);
glShadeModel(GL_FLAT);
}
```

```
void display(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
drawScene ();
selectObjects ();
glFlush();
}


int main(int argc, char** argv)
{
auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
auxInitPosition (0, 0, 200, 200);
auxInitWindow (argv[0]);
myinit ();
auxMainLoop(display);
}
```

16.4 拾　　取

作为 16.3 节所描述的处理过程的一个扩展，可以用选择模式确定物体是否被拾取了。要做到这一点，用一个与投影矩阵相关联的拾取矩阵，将绘制限制在视口的一个很小的区域里，通常是在光标附近。然后允许某种形式的输入，如单击鼠标，引发选择模式的初始化。建立起选择模式并使用特定的拾取矩阵，绘制在光标附近的物体引发选择命中。即，通常在拾取的时候确定哪个物体被画在光标附近了。

16.4.1 简单示例

拾取的建立几乎与正规的选择模式完全一样，只有以下几点主要区别：

拾取通常由某个输入设备触发。在下面的例子中，按单击鼠标运行实现拾取的函数。用例程 gluPickMatrix()将一个特殊的投影矩阵乘到当前矩阵上。它要在把投影矩阵乘到栈上之前调用。

实现拾取的完全不同的方法在前面我们已经有所描述。这一技术使用颜色值标识物体的不同部件。

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble
        height, GLint viewport[4]);
```

建立一个投影矩阵，用于将绘制限制在视口的一个小区域里，并将这个矩阵乘到当前矩阵栈上。拾取区域的中心是窗口坐标(x，y)处，通常是光标位置。width 和 height 定义选取区域大小，用屏幕坐标。(你可以认为宽和高是设备第三的)。viewport[]表明当前视口边界，这可以用调用 glGetIntegerv(GL_VIEWPORT，GLint *viewport);获得。

深入 gluPickMatrix()建立的矩阵的净结果(net result)是将裁剪区域变换到单位立方体-1≤(x，y，z)≤ 1(或-w≤(wx，wy，wz)≤w)。拾取矩阵实际上提供了一个正交变换。将这个单位立方体的一个子区域映射到单位立方体。因为变换是任意的，你可以让拾取为不同类型的领域工作，如，用于旋转窗口的矩形区域。在某些情况下，你会发觉用附加的裁剪平面定义拾取区域要容易些。

例 16.3 演示了一个简单的选取。也演示了怎样用多个名字标识同一图元的不同部件，在这里行和列是一个被选物体。一个 3×3 矩阵被画出来，每个广场有不同的颜色。数组 board[3][3]维护每个块的当前单击总数。当前鼠标左键被按下，pickSquares()例子被调用来识别哪个方块被鼠标拾取了。在网格中 2 个名字标识一个方块：一个表示行，一个表示列。当单击鼠标，光标位置下的所有方块的颜色也会改变。

例 16.3 拾取实例 picksquare.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

int board[3][3]; /* 每一个方块中所用的颜色板 */

/* 为方块中的每一个小格设置清除颜色 */
void myinit(void)
{
int i, j;
for (i = 0; i < 3; i++)
for (j = 0; j < 3; j ++)
board[i][j] = 0;
glClearColor (0.0, 0.0, 0.0, 0.0);
}

void drawSquares(GLenum mode)
{
GLuint i, j;
for (i = 0; i < 3; i++) {
if (mode == GL_SELECT)
glLoadName (i);
for (j = 0; j < 3; j ++) {
if (mode == GL_SELECT)
```

```
glPushName (j);
glColor3f ((GLfloat) i/3.0, (GLfloat) j/3.0,
(GLfloat) board[i][j]/3.0);
glRecti (i, j, i+1, j+1);
if (mode == GL_SELECT)
glPopName ();
}
}
}

void processHits (GLint hits, GLuint buffer[])
{
unsigned int i, j;
GLuint ii, jj, names, *ptr;

printf ("hits = %d\n", hits);
ptr = (GLuint *) buffer;
for (i = 0; i < hits; i++) { /* 对每一次命中 */
names = *ptr;
printf (" number of names for this hit = %d\n", names);
ptr++;
printf (" z1 is %u;", *ptr); ptr++;
printf (" z2 is %u\n", *ptr); ptr++;
printf (" names are ");
for (j = 0; j < names; j++) { /* 对每一个名称 */
printf ("%d ", *ptr);
if (j == 0) /* 设置行和列 */
ii = *ptr;
else if (j == 1)
jj = *ptr;
ptr++;
}
printf ("\n");
board[ii][jj] = (board[ii][jj] + 1) % 3;
}
}
#define BUFSIZE 512
void pickSquares(AUX_EVENTREC *event)
{
```

```
GLuint selectBuf[BUFSIZE];
GLint hits;
GLint viewport[4];
int x, y;

x = event->data[AUX_MOUSEX];
y = event->data[AUX_MOUSEY];
glGetIntegerv (GL_VIEWPORT, viewport);

glSelectBuffer (BUFSIZE, selectBuf);
(void) glRenderMode (GL_SELECT);

glInitNames();
glPushName(-1);

glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
/* 在鼠标附近创建 5×5 个象素的拾取区 */
gluPickMatrix((GLdouble) x,
(GLdouble) (viewport[3] - y), 5.0, 5.0, viewport);
gluOrtho2D (0.0, 3.0, 0.0, 3.0);
drawSquares (GL_SELECT);
glPopMatrix ();
glFlush ();

hits = glRenderMode (GL_RENDER);
processHits (hits, selectBuf);
}

void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
drawSquares (GL_RENDER);
glFlush();
}
void myReshape(GLsizei w, GLsizei h)
{
glViewport(0, 0, w, h);
```

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D (0.0, 3.0, 0.0, 3.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

int main(int argc, char** argv)
{
auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
auxInitPosition (0, 0, 100, 100);
auxInitWindow (argv[0]);
myinit ();
auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, pickSquares);
auxReshapeFunc (myReshape);
auxMainLoop(display);
}
```

16.4.2 从多个名称和一个层次结构中进行拾取

多个名称也可以用来在场景的层次结构物体中进行选择。例如，如果正在渲染一个汽车的一组线，可能希望用户移动鼠标拾取第 3 辆汽车上左前方车轮上的第 3 个螺钉，可以使用不同的名称来标识出每个层次：哪一辆车、哪一个车轮以及哪一个螺钉。作为另外一个例子，一个名称可以用来描述在其他分子中间的一个单一分子，附加的名称可以区分在这个分子中的一个原子。

例 16.4 创建多个名称

```
draw_wheel_and_bolts()
{
long i;

draw_wheel_body();
for (i = 0; i < 5; i++) {
glPushMatrix();
glRotate(72.0*i, 0.0, 0.0, 1.0);
glTranslatef(3.0, 0.0, 0.0);
glPushName(i);
draw_bolt_body();
glPopName();
glPopMatrix();
```

```
}
}

draw_body_and_wheel_and_bolts()
{
draw_car_body();
glPushMatrix();
glTranslate(40,  0,  20); /* 第 1 个车轮的位置*/
glPushName(1); /* 1 号车轮的名称 */
draw_wheel_and_bolts();
glPopName();
glPopMatrix();
glPushMatrix();
glTranslate(40,  0,  .20); /* 第 2 号车轮的位置 */
glPushName(2); /* 2 号车轮的名称*/
draw_wheel_and_bolts();
glPopName();
glPopMatrix();

/* 类似地绘制另外两个车轮 */
}
```

例 16.5 使用例 16.4 中的程序绘制三辆不同的汽车，分别编号为 1，2，3

```
draw_three_cars()
{
glInitNames();
glPushMatrix();
translate_to_first_car_position();
glPushName(1);
draw_body_and_wheel_and_bolts();
glPopName();
glPopMatrix();

glPushMatrix();
translate_to_second_car_position();
glPushName(2);
draw_body_and_wheel_and_bolts();
glPopName();
glPopMatrix();
glPushMatrix();
```

```
translate_to_third_car_position();
glPushName(3);
draw_body_and_wheel_and_bolts();
glPopName();
glPopMatrix();
}
```

假设已经进行了拾取，以下就是一些可能的名称堆栈的返回值和对它们的解释。在这些例子中，至多会有一个命中记录返回，其中 d1，d2 都是深度值。当返回空值时表明拾取动作发生在所有汽车之外。

```
2 d1d2 2 1 Car 2,  wheel 1
1 d1d2 3 Car 3 body
3 d1d2 1 1 0 Bolt 0 on wheel 1 on car 1
```

最后一个解释中，假设螺钉和车轮不占据相同的拾取区域。用户可能同时拾取了车轮和螺钉，那么将产生 2 次命中，必须判断处理哪一个命中，也许通过使用深度值可以判定哪一个被拾取的物体离视点最近。下一节中将进一步研究深度值在拾取中的使用。

例 16.6 演示了如何在拾取时使用深度值判定被拾取的是哪一个物体。这个程序先在普通绘制模式下绘制了 3 个相互遮挡的矩形。当单击鼠标时，调用 pickRects()函数。程序将首先返回鼠标的位置，然后进入选择模式，初始化名称堆栈，然后在设置正交投影矩阵之前将拾取矩阵乘到矩阵栈的栈顶元素上去。当在某个矩形上单击鼠标时，就会发生一次选择命中。最后，检查选择缓存的内容来判断哪一个被命名了的对象离拾取区域最近。

例 16.6　带深度值的拾取 pickdepth.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void myinit(void)
{
glClearColor (0.0,  0.0,  0.0,  0.0);
glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);
glShadeModel(GL_FLAT);
glDepthRange (0.0,  1.0); /* 默认区映射 */
}

void drawRects(GLenum mode)
{
if (mode == GL_SELECT)
glLoadName (1);
glBegin (GL_QUADS);
```

```
glColor3f (1.0, 1.0, 0.0);
glVertex3i (2, 0, 0);
glVertex3i (2, 6, 0);
glVertex3i (6, 6, 0);
glVertex3i (6, 0, 0);
glColor3f (0.0, 1.0, 1.0);
glVertex3i (3, 2, -1);
glVertex3i (3, 8, -1);
glVertex3i (8, 8, -1);
glVertex3i (8, 2, -1);
glColor3f (1.0, 0.0, 1.0);
glVertex3i (0, 2, -2);
glVertex3i (0, 7, -2);
glVertex3i (5, 7, -2);
glVertex3i (5, 2, -2);
glEnd ();
}

void processHits (GLint hits, GLuint buffer[])
{
unsigned int i, j;
GLuint names, *ptr;

printf ("hits = %d\n", hits);
ptr = (GLuint *) buffer;
for (i = 0; i < hits; i++) { /* 对于每一次命中 */
names = *ptr;
printf (" number of names for hit = %d\n", names);
ptr++;
printf (" z1 is %u;", *ptr); ptr++;
printf (" z2 is %u\n", *ptr); ptr++;
printf (" the name is ");
for (j = 0; j < names; j++) { /* 对于每一个名称 */
printf ("%d ", *ptr); ptr++;
}
printf ("\n");
}
}
```

```
#define BUFSIZE 512

void pickRects(AUX_EVENTREC *event)
{
GLuint selectBuf[BUFSIZE];
GLint hits;
GLint viewport[4];
int x, y;

x = event->data[AUX_MOUSEX];
y = event->data[AUX_MOUSEY];
glGetIntegerv (GL_VIEWPORT, viewport);

glSelectBuffer (BUFSIZE, selectBuf);
(void) glRenderMode (GL_SELECT);

glInitNames();
glPushName(-1);

glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
/* 在鼠标附近创建 5×5 像素的拾取区 */
gluPickMatrix ((GLdouble) x,
(GLdouble) (viewport[3] - y), 5.0, 5.0, viewport);
glOrtho (0.0, 8.0, 0.0, 8.0, 0.0, 2.0);
drawRects (GL_SELECT);
glPopMatrix ();
glFlush ();

hits = glRenderMode (GL_RENDER);
processHits (hits, selectBuf);
}

void display(void)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
drawRects (GL_RENDER);
glFlush();
```

```
}
void myReshape(GLsizei w, GLsizei h)
{
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho (0.0, 8.0, 0.0, 8.0, 0.0, 2.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
int main(int argc, char** argv)
{
auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
auxInitPosition (0, 0, 100, 100);
auxInitWindow (argv[0]);
myinit ();
auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, pickRects);
auxReshapeFunc (myReshape);
auxMainLoop(display);
}
```

第 17 章　用 OpenGL 编写屏幕保护程序

许多人都欣赏过 Windows 环境下 OpenGL 编写的屏幕保护程序。用户是否惊叹过这些三维图形的变幻莫测？是否因此产生了无穷遐想？是否因编写者的奇特创意而拍手叫绝？学习完本章后，用户可以尽情发挥聪明才智，将奇思异想付诸实际。也可以将前面讲解的一些漂亮的例子改写为屏幕保护程序，送给朋友或自己使用，一定会其乐无穷。

本章先介绍 Windows 环境下屏幕保护的一般原理及设置方法，然后，讲解编写屏幕保护程序的基本步骤，最后，通过一个具体的屏幕保护程序编写示例让读者产生更深刻的印象，从而得心应手地编写自己的屏幕保护程序。

本章的内容可视为 OpenGL 高级编程的一个重要补充，因为 OpenGL 作为一个功能强大的图形绘制编程工具，它在桌面娱乐方面的应用也不容忽视。

17.1　Windows 环境下的屏幕保护程序

屏幕保护程序是 Windows 环境下的一种重要的应用程序，它不仅在人们长时间不工作时保护了屏幕和硬盘数据(因为它可以设置密码保护)，还通过各种美妙的图形美化了人们的视野。在喝茶或读报的时候，偶尔瞥一眼机器屏幕，发现上面正悄然运行着屏幕保护程序，一定会有说不出的安全感和舒适感。

17.1.1　设置屏幕保护程序

Windows 下屏幕保护程序的设置非常简单，可以按以下步骤设置：

(1) 单击鼠标右键，弹出如图 17.1 所示的菜单。

(2) 单击【属性】菜单项，弹出如图 17.2 所示的对话框。

(3) 单击【屏幕保护程序】标签，显示【屏幕保护程序】选项卡，如图 17.3 所示。

(4) 打开【屏幕保护程序】下拉列表框，从中选择喜欢的屏幕保护程序，在窗口中部的显示器中可以看到屏幕保护程序运行的大致情形。例如，用 OpenGL 绘制的三维管道作为屏幕保护程序时，窗口的内容如图 17.4 所示。

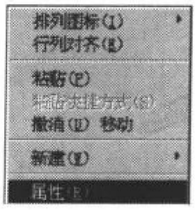

图17.1　桌面窗口右键菜单

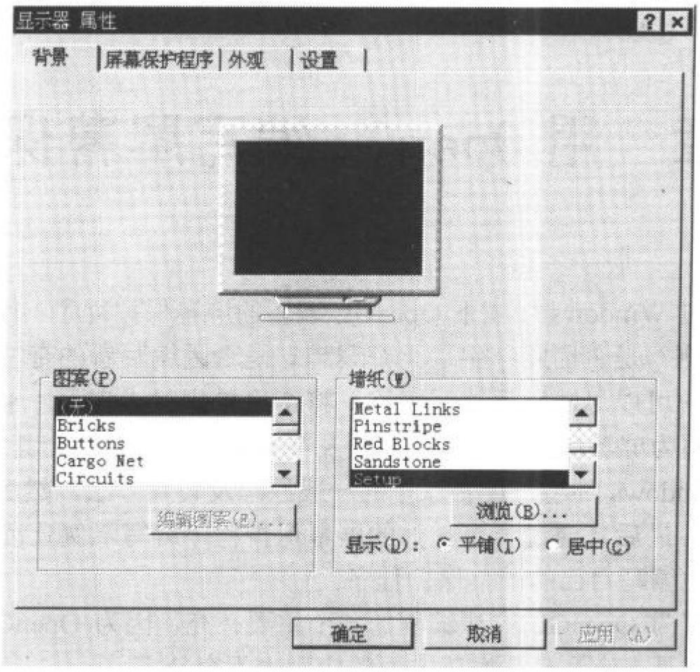

图17.2 【显示器属性】对话框

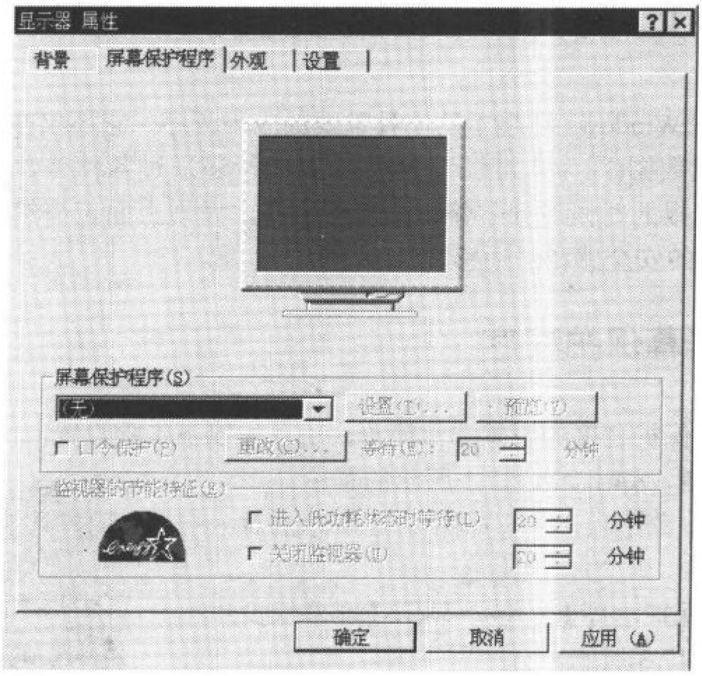

图17.3 【屏幕保护程序】选项卡

此外，还可以设置口令保护，从而在暂时离开时避免别人随便打开计算机，从而起到保护作用。单击【设置】按钮，可以设置屏幕保护程序运行时所需的一些必要的状态和参数信息，通常不同的屏幕保护程序提供不同的设置窗口，这是因为设置窗口是屏幕保护程序的一部分，而并非设置程序的一部分。例如，三维管道屏幕保护程序的设置程序如图 17.5 所示，显然并不是每个程序都需要这些信息。

图17.4　选择屏幕保护程序

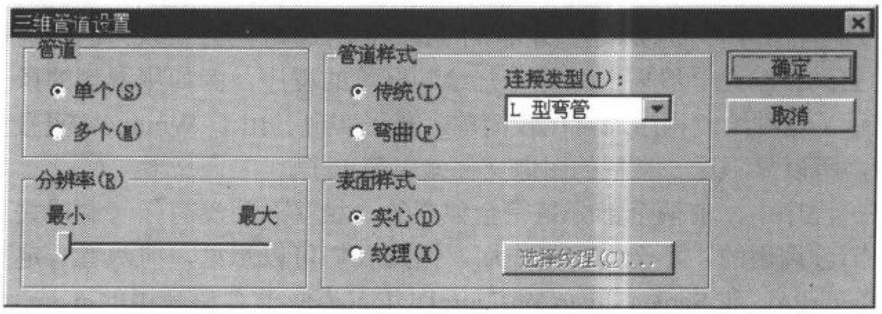

图17.5　三维管道屏幕保护程序设置窗口

还可以设置屏幕保护程序的启动时间间隔。希望在启动屏幕保护程序之前对它有一个完整的认识，还可以单击【预览】按钮观看屏幕保护程序的运行结果。此时的运行结果和真正启用屏幕保护程序时是完全一样的，包括屏幕的显示内容和密码保护等。

屏幕保护程序事实上是以.scr 为后缀的可执行文件，若将文件后缀改为.exe，程序也能运行，不过这时只能显示出设置窗口。

注意： *并不是所有的.scr 可执行文件都可以改成屏幕保护程序，只有指定了屏幕保护事件响应的程序才能成为正确的屏幕保护程序。*

17.1.2　安装屏幕保护程序

屏幕保护程序和其他应用程序一样可以通过安装程序安装，安装完毕后将会在系统注册表里添加自动这一屏幕保护程序，当希望从列表中删除这个屏幕保护程序时，只要在控制面板里选择删除该程序即可。由于屏幕保护程序实质上就是 Windows 目录下的以.scr 为后缀的文件，因此也可以直接将想要的以.scr 为扩展名的屏幕保护程序拷贝到 Windows 目

录下。不过，用这种方法安装的屏幕保护程序不能通过控制面板的【添加/删除程序】来卸载。

安装好(或拷贝到 Windows 目录下)之后，执行上面介绍的设置屏幕保护程序的方法，会发现在屏幕保护程序下拉列表框中增加了新安装的屏幕保护程序。

17.2 屏幕保护程序编写的基本步骤

通过使用屏幕保护程序知道，一个完整的屏幕保护程序由 3 部分组成：

- 屏幕保护响应程序

这是一个主程序，用于在计算机长时间没有使用时执行程序。

- 屏幕保护设置程序

这一部分在设置屏幕保护参数时使用。有两种设置方法，一种是在屏幕保护选择窗口中单击【设置】按钮，一种是在资源管理器中，用右键单击需要进行设置的屏幕保护程序，在弹出的设置窗口中进行设置，效果是一样的。

- 屏幕绘制程序

最简单的绘制程序是黑屏，即什么也不绘制，只清除屏幕的原有内容，这种方法同样可以起到保护屏幕的作用，但美观性要欠缺得多，而且有时还会误以为已经关机而切断电源造成数据的丢失。因此一般需要编写一些绘制屏幕的程序，例如用漂亮的风景图片或美女像交替地显示。这种绘制程序通常比较简单，易于实现。由于 Windows 直接提供了对位图图像的支持，所以显示这一类型的图像文件就更简单了。

在屏幕保护程序中，前两个部分是完全独立的，相互之间没有任何调用关系，而第三部分是被第一部分调用的，不会自己起作用。因此我们可以想见，前两者一定有一个专门的名称 screenSaverProc 和 ScreenSaverConfigureDialog(类似于 C 语言中的 main 函数)，而第三部分则无需指定特定的函数名称。

17.2.1 屏幕保护响应程序

屏幕保护响应程序是当长时间没有操作时，计算机自动响应而调用的函数，它是整个屏幕保护程序的入口。一般来说，它是一个名称为 ScreenSaverProc 的函数，函数的原型定义在 VC 的 include 目录下的 scrnsave.h 文件中。函数原型如下：

```
LRESULT WINAPI ScreenSaverProc (HWND hWnd, UINT message,
                               WPARAM wParam, LPARAM lParam);
```

为了加强对屏幕保护程序工作原理的理解，把 scrnsave.h 文件的内容列在下面，这样，对于每个函数的原型和使用都有了较为详细的说明。

```
/*****************************************************************
              **************/
* scrnsave.h Windows NT 3.1 screensaver defines and definitions.*
*Version 1.0*
```

```
*NOTE: windows.h must be #included first*
*Windows NT NOTE:(Differences from Win 3.1 Screensavers)*
*All Screensavers are required to have a Description string*
*of no more than 25 chars for display by the Control Panel's*
*Desktop applet. This is string 1 in the resource string*
*table of the Windows 32.bit screen saver .SCR (.EXE) file.*
*Passwords for Windows NT Screen Savers are handled by the*
*Winlogon process. If the registry value:*
*HKEY_CURRENT_USER\Control Panel\Desktop\ScreenSaverIsSecure*
*is nonzero, Winlogon will ask for the User's login password*
*before allowing the Screen Saver to exit. All password data*
*and dialogs have been removed from individual Screensavers.*
*Copyright (c) 1992.1996, Microsoft Corp. All rights reserved.*
/*************************************************************
            **************/
#ifndef _INC_SCRNSAVE
#define _INC_SCRNSAVE
#include <pshpack1.h>   /* Assume byte packing throughout */

#ifdef __cplusplus
extern "C" {          /* Assume C declarations for C++ */
#endif      /* __cplusplus */

/* MANDATORY string required in .RC file
 * This string should contain a less than 25 char name/description
          of the
 * screen saver. This string is what will be seen by the user in the
          Control
 * Panel's Desktop applet screen saver listbox.
 */
#define IDS_DESCRIPTION     1
/* Icon resource ID.
 *
 * This should be the first icon used and must have this resource
          number.
 * This is needed as the first icon in the file will be grabbed
 */
#define ID_APP     100
#define DLG_SCRNSAVECONFIGURE   2003
```

```
#define idsIsPassword        1000
#define idsIniFile           1001
#define idsScreenSaver       1002
#define idsPassword          1003
#define idsDifferentPW       1004
#define idsChangePW          1005
#define idsBadOldPW          1006
#define idsAppName           1007
#define idsNoHelpMemory      1008
#define idsHelpFile          1009
#define idsDefKeyword        1010

/* This function is the Window Procedure for the screen saver.  It
        is
 * up to the programmer to handle any of the messages that wish to
        be
 * interpretted.  Any unused messages are then passed back to
 * DefScreenSaverProc if desired which will take default action on
        any
 * unprocessed message...
 */
#ifdef UNICODE
LRESULT WINAPI ScreenSaverProcW (HWND hWnd, UINT message, WPARAM
        wParam, LPARAM lParam);
#  define  ScreenSaverProc ScreenSaverProcW
#else
LRESULT WINAPI ScreenSaverProc  (HWND hWnd, UINT message, WPARAM
        wParam, LPARAM lParam);
#endif
/* This function performs default message processing.  Currently
        handles
 * the following messages:
 *
 * WM_SYSCOMMAND:   return FALSE if wParam is SC_SCREENSAVE or
      • SC_CLOSE
 *
 * WM_DESTROY:      PostQuitMessage(0)
 *
```

```
 * WM_SETCURSOR:   By default, this will set the cursor to a null
           cursor,
 *               thereby removing it from the screen.
 *
 * WM_LBUTTONDOWN:
 * WM_MBUTTONDOWN:
 * WM_RBUTTONDOWN:
 * WM_KEYDOWN:
 * WM_KEYUP:
 * WM_MOUSEMOVE:    By default, these will cause the program to
           terminate.
 * Unless the password option is enabled.  In that case
 * the DlgGetPassword() dialog box is brought up.
 *
 * WM_NCACTIVATE:
 * WM_ACTIVATEAPP:
 * WM_ACTIVATE:     By default, if the wParam parameter is FALSE
           (signifying
 * that transfer is being taken away from the application),
 * then the program will terminate.  Termination is
 * accomplished by generating a WM_CLOSE message.  This way,
 * if the user sets something up in the WM_CREATE, a
 * WM_DESTROY will be generated and it can be destroyed
 * properly.
 * This message is ignored, however is the password option
 * is enabled.
 */
LRESULT WINAPI DefScreenSaverProc (HWND hWnd, UINT msg, WPARAM wParam,
           LPARAM lParam);
/* A function is also needed for configuring the screen saver.  The
           function
 * should be exactly like it is below and must be exported such that
           the
 * program can use MAKEPROCINSTANCE on it and call up a dialog box.
           Further.
 * more, the template used for the dialog must be called
 * ScreenSaverConfigure to allow the main function to access it...
 */
BOOL WINAPI ScreenSaverConfigureDialog (HWND hDlg, UINT message,
```

```
              WPARAM wParam, LPARAM lParam);
/* To allow the programmer the ability to register child control
              windows, this
 * function is called prior to the creation of the dialog box.  Any
 * registering that is required should be done here, or return TRUE
              if none
 * is needed...
 */
BOOL WINAPI RegisterDialogClasses (HANDLE hInst);
/* The following functions are called by DefScreenSaverProc and must
 * be exported by all screensavers using this model.
 */
/*
 * There are only three other points that should be of notice:
 * 1) The screen saver must have a string declared as 'szAppName'
              contaning the
 * name of the screen saver, and it must be declared as a global.
 * 2) The screen saver EXE file should be renamed to a file with a
              SCR
 * extension so that the screen saver dialog from the control panel
              can
 * find it when is searches for screen savers.
 */
#define WS_GT   (WS_GROUP | WS_TABSTOP)
#define MAXFILELEN        13
#define TITLEBARNAMELEN    40
#define APPNAMEBUFFERLEN   40
#define BUFFLEN         255
/* The following globals are defined in scrnsave.lib */
extern HINSTANCE hMainInstance;
extern HWND   hMainWindow;
extern BOOL   fChildPreview;
extern TCHAR  szName[TITLEBARNAMELEN];
extern TCHAR  szAppName[APPNAMEBUFFERLEN];
extern TCHAR  szIniFile[MAXFILELEN];
extern TCHAR  szScreenSaver[22];
extern TCHAR  szHelpFile[MAXFILELEN];
extern TCHAR  szNoHelpMemory[BUFFLEN];
extern UINT   MyHelpMessage;
```

```
/* OPTIONAL . Win95 Only */
#define SCRM_VERIFYPW   WM_APP
/*
 * This message is sent to the main screen saver window when password
 * protection is enabled and the user is trying to close the screen
          saver.  You
 * can process this message and provide your own validation technology.
          If you
 *  process  this  message,  you  should  also  support  the
          ScreenSaverChangePassword
 * function, described below.  Return zero from this message if the
          password
 * check failed.  Return nonzero for success.  If you run out of memory
          or
 * encounter a similar class of error, return non.zero so the user
          isn't left
 * out in the cold.  The default action is to call the Windows Master
 * Password Router to validate the user's password.
 */
void WINAPI ScreenSaverChangePassword( HWND hParent );
/*
 * You supply this if you provide your own authentication.  Windows
          will call
 * it when the user wants to change the password.  An implementation
          of this
 * function should present password change UI to the user.
 *  You  should  only  supply  this  function  if  you  also  hook  the
          SCRM_VERIFYPW
 * message to validate passwords.
 * The default action is to call the Windows Master Password Router.
 */
#ifdef __cplusplus
}
#endif      /* __cplusplus */
#include <poppack.h>
#endif  /* !_INC_SCRNSAVE */
```

17.2.2 屏幕保护设置程序

屏幕保护设置程序的函数原型在前面的 scrnsave.h 中已经给出，这里不再赘述。此函数的作用在于处理屏幕保护设置对话框中的各种信息，例如 OK，Cancel 等按钮的处理等。当对话框中还有其他事件需要处理时，同样可以在此函数中进行。例如，一个简单的设置程序可以是简单地显示一个对话框，而勿作任何参数设置，这时可以编写程序如下：

```
BOOL CALLBACK ScreenSaverConfigureDialog(HWND hWnd, UINT Msg, WPARAM
            wParam, LPARAM lParam)
{
switch(Msg)
{
case WM_COMMAND:
switch(LOWORD(wParam))
{
case IDOK:  // Exit the dialog
case IDC_EXIT:
EndDialog(hWnd, 42);
break;
}
break;
default:
return FALSE;
break;
}
return TRUE;
}
```

上面的程序只进行了窗口中两个按钮的处理。需要处理其他信息时，只要相应地增加信息处理程序即可。

17.2.3 图形绘制程序

图形绘制程序是屏幕保护程序的主要部分，在其中实现了屏幕内容的显示。通常屏幕保护程序的图形绘制都不是静态的显示过程，而是动态的显示过程，即图形绘制的内容随着时间的推移变换。通过在 ScreenSaverProc 函数中增加定时事件处理可以实现图形绘制。

为了用 OpenGL 产生绘制程序，可以在 WM_CREATE 的事件处理程序中对 OpenGL 进行初始化。为此，可以编写下面这段程序：

```
LONG CALLBACK ScreenSaverProc(HWND hWnd, UINT msg, WPARAM wParam,
            LPARAM lParam)
```

```
{
            RECT rect;
            switch (msg) {
            case WM_CREATE:
            ghDC = GetDC(hWnd);
      if (!bSetupPixelFormat(ghDC))
         PostQuitMessage (0);
      ghRC = wglCreateContext(ghDC);
      wglMakeCurrent(ghDC, ghRC);
      GetClientRect(hWnd, &rect);
      initializeGL(rect.right, rect.bottom);
      resize(rect.right, rect.bottom);
           break;
           case WM_TIMER:
      // 显示一个方框
           DrawFrame(hWnd);
           return 0;
  case WM_ERASEBKGND:
      wTimer = SetTimer(hWnd, 0, 50, NULL);
           break;
  case WM_DESTROY:           /* 销毁窗口事件 */
           if (ghRC)
         wglDeleteContext(ghRC);
     if (ghDC)
         ReleaseDC(hWnd, ghDC);
           if (wTimer) // 如果已创建了定时器，此时关闭它。
               KillTimer(hWnd, wTimer);
           break;
  }
  return DefScreenSaverProc(hWnd, msg, wParam, lParam);
}
```

可以这样描述这段程序，即：首先在 WM_CREATE 事件处理程序中对 OpenGL 进行初始化，然后在 WM_TIMER 事件(即定时事件)处理程序中调用函数绘制一帧画面，在 WM_ERASEBKGND 事件处理程序中设置定时器事件，最后，在 WM_DESTROY 事件处理程序中删除 OpenGL 绘制描述表和 Windows 绘图描述表定时器。

在真正的图形绘制程序中，可以按以往的 OpenGL 绘制程序一样进行编程。需要注意的是，要根据具体情况设置对话框的参数信息或定时器的时间信息，从而进行相应的编程。

17.3 屏幕保护程序实例

本节将通过一个屏幕保护程序实例介绍屏幕保护程序的编程步骤。为了简单起见，不对屏幕保护设置程序进行复杂处理，只简单地显示一个窗口，单击 Exit 按钮程序返回。

17.3.1 屏幕保护程序主程序

编写屏幕保护程序所需的源代码如下：

```
#include <windows.h>
#include <stdlib.h>
#include <scrnsave.h>
#include "resource.h"

HANDLE hInst;        // 当前实例
UINT wTimer = 0;     // 定时器的 ID

extern HWND  ghWnd;
extern HDC   ghDC;
extern HGLRC ghRC;

/*
 * RegisterDialogClasses 函数是注册窗口类的入口点，它将被设置对话框调用 */
BOOL WINAPI RegisterDialogClasses(HINSTANCE hInst)
{
   return 1;
}

/*
 * ScreenSaverConfigureDialog 函数处理屏幕保护设置消息 */
BOOL CALLBACK ScreenSaverConfigureDialog(HWND hWnd, UINT Msg, WPARAM
            wParam, LPARAM lParam)
{
   switch(Msg)
   {
      case WM_COMMAND:
            switch(LOWORD(wParam))
         {
```

```
                case IDOK: // 退出对话框
                case IDC_EXIT:
                        EndDialog(hWnd, 42);
                        break;
            }
        break;
        default:
               return FALSE;
        break;
    }
    return TRUE;
}

/*****************************************************************
          ************
   FUNCTION: ScreenSaverProc(HWND, UINT, WPARAM, LPARAM)
   目的: 处理屏幕保护消息
******************************************************************
          ***********/
LONG CALLBACK ScreenSaverProc(HWND hWnd, UINT msg, WPARAM wParam,
          LPARAM lParam)
{
          RECT rect;
          switch (msg) {
          case WM_CREATE:

          ghDC = GetDC(hWnd);
      if (!bSetupPixelFormat(ghDC))
         PostQuitMessage (0);

      ghRC = wglCreateContext(ghDC);
      wglMakeCurrent(ghDC, ghRC);
      GetClientRect(hWnd, &rect);
          initializeGL(rect.right, rect.bottom);
      resize(rect.right, rect.bottom);
          break;
          case WM_TIMER:
      // 显示窗体中的动画
          DrawFrame(hWnd);
```

```
            return 0;
    case WM_ERASEBKGND:
            // 产生一个随机数种子
        srand(GetTickCount());
            // 指定窗口程序 50 毫秒发生一个定时器消息
        wTimer = SetTimer(hWnd, 0, 10, NULL);
            break;
    case WM_DESTROY:              /* 窗口销毁消息*/
            if (ghRC)
          wglDeleteContext(ghRC);
        if (ghDC)
          ReleaseDC(hWnd, ghDC);
            if (wTimer) // 如果已创建了定时器此时使其停止
              KillTimer(hWnd, wTimer);
            break;
    }
    return DefScreenSaverProc(hWnd, msg, wParam, lParam);
}
```

其中，创建的屏幕保护设置对话框如图 17.6 所示。

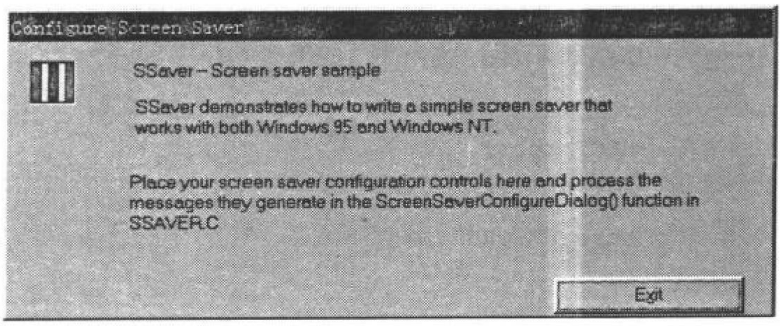

图17.6 屏幕保护设置窗口

另外，为了使编程结果成为屏幕保护程序，必须打开 Project 的 Settings 菜单，在弹出的环境设置对话框中单击 Link 标签，将 Output file name 设置为以.scr 为后缀的名称。值得注意的是在 VC 环境下运行该程序，只会产生上述屏幕保护设置程序的结果。不过，在脱离 VC 环境下运行即可得到正确的结果，因此不必担心。另外，如果想在资源管理器下运行设置程序，可以通过在该文件上单击右键来实现。

17.3.2 图形绘制程序

这里绘制的内容是一系列用 gluSphere(qobj,1.0,20,20)函数生成的球体，球的颜色是由随机数产生的，新球的产生位置由定时函数控制。图形绘制的结果如图 17.7 所示。

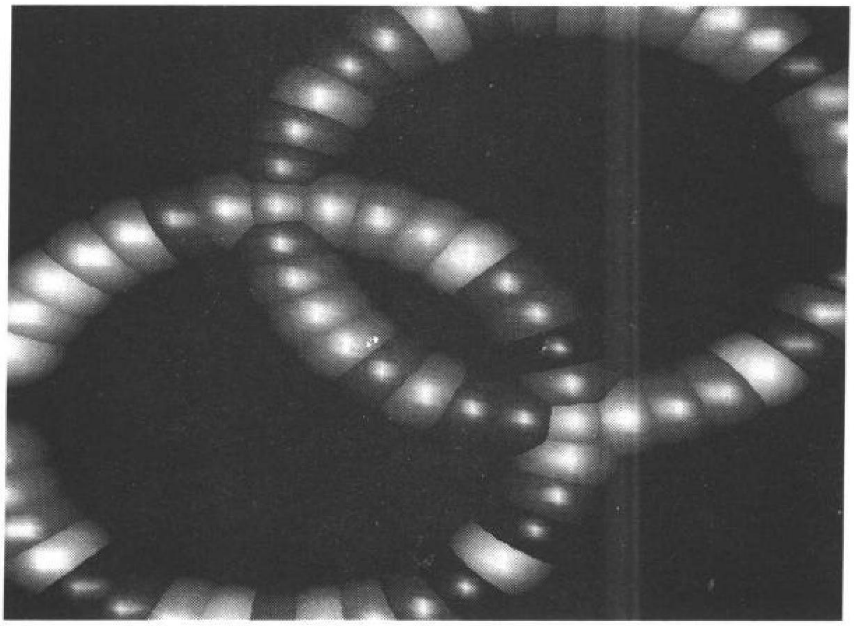

图17.7　屏幕保护程序图形绘制结果

程序代码如下所示：

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>

#define     DEG_TO_RAD  (3.1415927 / 180.0)
#define     BOXWIDTH    100.0

GLfloat     azimuth = 0.0;
GLfloat     inclin = 0.0;

/* 程序中的全局变量和函数原型 */
CHAR szAppName[]="View";
HWND  ghWnd;
HDC   ghDC;
HGLRC ghRC;

GLUquadricObj *qobj;

#define WIDTH   600
#define HEIGHT  600
```

```
BOOL bSetupPixelFormat(HDC);

GLvoid resize(GLsizei, GLsizei);
GLvoid initializeGL(GLsizei, GLsizei);
GLvoid drawScene(GLvoid);
void initLighting(void);
void setTransform(void);
void DrawFrame(HWND hWnd);

GLint ell;
GLfloat dist=0.0;

GLvoid initializeGL(GLsizei width, GLsizei height)
{
            initLighting();

            glEnable(GL_NORMALIZE);
   glClearColor(0.0, 0.0, 0.0, 0.0);
}

BOOL bSetupPixelFormat(HDC hdc)
{
   PIXELFORMATDESCRIPTOR *ppfd;
   int pixelformat;

PIXELFORMATDESCRIPTOR pfd =
       sizeof(PIXELFORMATDESCRIPTOR),  // 结构的大小
       1,                        // 版本号
       PFD_DRAW_TO_WINDOW |   // 支持家
       PFD_SUPPORT_OPENGL,    // 支持 OpenGL
       //PFD_DOUBLEBUFFER,    // 双缓存
       PFD_TYPE_RGBA,        // RGBA 模式
       24,                       // 24 位颜色深度
       0, 0, 0, 0, 0, 0,       // 忽略颜色位
       0,                        // 无 alpha 缓存
       0,                        // 移位位忽略
       0,                        // 无累积位
       0, 0, 0, 0,               // 累积位
       32,                       // 32 位深度缓存
```

```
        0,                          // 无模板缓存
        0,                          // 无辅助缓存
        PFD_MAIN_PLANE,             // 主层
        0,                          // 保留
        0, 0, 0                     // 忽略层掩模
    };

    pfd.cColorBits = GetDeviceCaps(ghDC,BITSPIXEL);

        ppfd = &pfd;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error",
            MB_OK);
        return FALSE;
    }
    if (SetPixelFormat(hdc, pixelformat, ppfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error",
            MB_OK);
        return FALSE;
    }
    return TRUE;
}

/* OpenGL 代码 */
GLvoid resize( GLsizei width, GLsizei height )
{

    glViewport( 0, 0, width, height );

            setTransform();
    drawScene();
}

static void  spherToRect (double azim, double inc, double
```

```
                disteye,
    double *px, double *py, double *pz)
{
    double theta, phi, rho; /* 球坐标 */

    theta = azim - 90.0; /* theta 从+X 轴开始 */
    phi = 90.0 - inc; /* phi 变化范围是 0~180 */
    rho = disteye;

    *px = rho * sin(phi*DEG_TO_RAD) * cos(theta*DEG_TO_RAD);
    *py = rho * sin(phi*DEG_TO_RAD) * sin(theta*DEG_TO_RAD);
    *pz = rho * cos(phi*DEG_TO_RAD);
}
void setTransform( void )
{
    GLfloat eyedistance = 300.0;
    GLfloat twist = 0.0;
    GLdouble fovy = 40.0;
    GLdouble aspect = 1.0;
    GLdouble znear, zfar;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    znear = eyedistance - (BOXWIDTH * 1.5);
    zfar = eyedistance + (BOXWIDTH * 1.5);
    gluPerspective(fovy, aspect, znear, zfar);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glTranslatef(0.0, 0.0, -eyedistance+dist);
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-(90.0 - inclin), 1.0, 0.0, 0.0);
    glRotated(-azimuth, 0.0, 0.0, 1.0);
}

static void initLighting( void )
{
    GLfloat light_position[] = { 0.0, 0.0, 0.0, 0.0 };
```

```
GLfloat    light_diffuse[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat    light_specular[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat    lmodel_ambient[] = { 0.0, 0.0, 0.0, 0.0 };
GLfloat    mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat    mat_shininess[] = { 50.0 };
double     litx, lity, litz;
static float  litazim = 0.0;
static float  litinclin = 0.0;

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);

spherToRect((double)litazim, (double)litinclin, 1.0,
            &litx, &lity, &litz);

/* 设置正确的坐标系：X 轴指向观察者，X 轴指向右方，Y 轴指向上方*/
light_position[1] = (GLfloat)litx;
light_position[2] = (GLfloat)litz;
light_position[3] = (GLfloat).lity;

glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lmodel_ambient);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,
        mat_specular);
glMaterialfv(GL_FRONT_AND_BACK,GL_SHININESS,
        mat_shininess);
glColorMaterial(GL_FRONT_AND_BACK, GL_DIFFUSE);
glEnable(GL_LIGHT0);
qobj = gluNewQuadric();
gluQuadricNormals(qobj,GLU_SMOOTH);
        ell = glGenLists(1);
        glNewList(ell,GL_COMPILE);
          gluSphere(qobj,1.0,20,20);
        glEndList();
        glClearColor(0.0, 0.0, 0.0, 0.0);
        glClear(
        GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|
```

```
                    GL_STENCIL_BUFFER_BIT);
}

static void DrawEllipsoid(void)
{
  float r = 20.0;
  glColor3f(1.0,1.0,0.0);
  glScalef(1.0,1.25,1.5);
  gluSphere(qobj,r,20,20);
  glScalef(1.0,1.0,1.0);
}

GLvoid drawScene(GLvoid)
{
static int theta = 0;
static int alpha = 0;
GLfloat x,y,z;
/* 设置屏幕清除色为黑色 */
if(!theta%360)
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|GL_STEN
            CIL_BUFFER_BIT);
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
theta = (theta + 10)%360;
glPushMatrix();
x = 100.0f*(GLfloat)cos(theta*DEG_TO_RAD) + 50.0f;
z = 100.0f*(GLfloat)sin(theta*DEG_TO_RAD) + 50.0f;
y = 100.0f;
glTranslatef(x,y,z);
            glColor3f(rand()%255/255.0f,rand()%255/255.0f,
            rand()%255/255.0f);
            glScalef(20.0f,20.0f,20.0f);
            glCallList(ell);
            glPopMatrix();
            glPushMatrix();
glTranslatef(-x,y,-z);
            glColor3f(rand()%255/255.0f,rand()%255/255.0f,
            rand()%255/255.0f);
            glScalef(20.0f,20.0f,20.0f);
```

```
                glCallList(ell);
                glPopMatrix();
                glPushMatrix();
 glTranslatef(-50.0f,0.0f,0.0f);
                glColor3f(rand()%255/255.0f,rand()%255/255.0f,
                rand()%255/255.0f);

                glRotatef((GLfloat)theta,1.0f,0.0f,0.0f);
                glScalef(25.0f,25.0f,25.0f);
                glPopMatrix();
                glFlush();

                glDisable(GL_LIGHTING);
                glDisable(GL_COLOR_MATERIAL);
}

void DrawFrame(HWND hWnd)
{
HDC hDC = GetDC(hWnd); // 得到窗口的设备描述表
    RECT rect;
    // 如果得不到正确的 DC 则返回
    if (!hDC)
        return;
                // 得到用于绘制的窗口的大小
    GetClientRect(hWnd, &rect);
    {
                setTransform();
                drawScene();
    }
    ReleaseDC(hWnd, hDC);
}
```

第 18 章　建筑物虚拟漫游编程

前面说过，图形学的三大发展方向是科学可视化、虚拟现实和真实感图形绘制，可以以 OpenGL 作为编程基础，实现这些高级应用。虚拟现实是近年来发展迅速的一项新兴技术。自从计算机发明以来，在传统的信息处理环境中一直以计算机为主体，是“人围着机器(计算机)转”的。例如在传统的仿真和建模环境中，虽然主观上一直在强调要发挥人的主动作用，但由于在客观上计算机只能在处理数字化的信息时才能发挥出强大的威力，人不得不去“凑合”当时的计算机所能提供的技术条件，人和计算机的关系是不甚和谐的。为了使计算机从只能处理数字化的单维信息转变为可以处理人所能感受到的、在思维过程中所接触到的、数字化信息之外的其他各种表现形式的多维信息，人们提出了虚拟现实的概念。

以下先对虚拟现实和虚拟漫游作简要的介绍，然后着重介绍建筑物内虚拟漫游的编程方法。

18.1　虚拟现实系统简介

虚拟现实(Virtual Reality)系统是一种可以创建和体验虚拟世界的计算机系统。虚拟世界是全体虚拟环境或给定仿真对象的全体。虚拟环境由计算机生成，通过视、听、触觉等作用于人，使人产生身临其境感觉的交互式视景仿真。研究如何设计和构成一个身临其境的虚拟现实系统，需要掌握计算机图形学、图像处理与模式识别、智能接口技术、人工智能技术、多传感器技术、语音处理与音响技术、网络技术、并行处理技术和高性能计算机系统方面的知识等，所有这些技术都是信息技术的分支。

18.1.1　虚拟现实技术的产生、基本特征及构成

目前世界正处在信息革命的时期，人们已开始认识到信息将是比物质资料和能源更为重要的劳动资料，事实上，它已经成为当今生产力发展中最活跃和最具有变革作用的劳动资料。在传统的信息处理环境中，由于技术条件的限制，处于主体地位的一直是计算机，人在其中的主观能动性很少得到发挥，这就形成了人和机器之间非常不和谐的关系。为了建立起和谐的人机环境，人们提出了多维信息空间的概念，即人与计算机的交流将不仅仅通过抽象的数字和文字表达，而且能够通过视觉、听觉、触觉和嗅觉等参与到信息环境中，从而使人产生身临其境的体验。虚拟现实技术将成为支持这个多维信息空间的关键技术。虚拟现实的英文名称是 Virtual Reality,Virtual 的英文本意是表现上具有真实事物的某些属性，但本质上是虚幻的。Reality 的英文本义是“真实”而不是“现实”。 从这个名字可

以看出，它英文本义是“真实世界的一个映像”(an image of real world)，而不仅仅只是一个人机界面。

从技术角度来说，这种适应于人的虚拟现实系统的特点可以简单地归纳为以下 3 个方面，即 3 个“I”，immersion-interaction-imagination(沉浸-交互-构想)。可以看出，3 个 I 强调了虚拟现实系统中人的主导作用。从过去人只能从计算机系统的外部去观测计算机处理的结果，到人能够沉浸到计算机系统所创建的环境中；从过去人只能通过键盘、鼠标与计算机环境中的单维数字化信息发生交互作用，到人能用多种传感器与多维化信息发生交互作用；从过去的人只能从以定量计算为主的结果中得到启发，从而加深对事物的认识，到人有可能从定性和定量综合集成的环境中得到感性和理性的认识，从而深化概念和萌发新意。创建一个虚拟现实系统的基本手段和目标就是利用并集成高性能的计算机软、硬件及各类先进的传感器，去创建一个使参与者处于一个具有身临其境的感觉、具有完善的交互作用能力、能帮助和启发构思的信息环境。

创建一个虚拟现实系统，在硬件方面需要以下几类设备的支持：

- 跟踪系统：确定参与者的头、手和身躯的位置；
- 触觉系统：提供力与压力的反馈；
- 音频系统：提供立体声源和判定空间位置；
- 图像生成和显示系统：产生可视化图像和立体显示；
- 高性能计算处理系统：具有高处理速度、大存储容量、强联网特性。

在软件方面，除了一般所需的软件支持外，主要是提供一个产生虚拟环境的工具集或产生虚拟环境的“外壳”。它至少应具有以下的功能：

- 能够接受各种高性能传感器的信息，如头盔的跟踪信息；
- 能生成立体的显示图形；
- 能把各种数据库(如地形地貌数据库、物体形象数据库等)、各种 CAD 软件进行调用和互联的集成环境。

18.1.2　多维信息空间

由于人类依靠自己的感知和认知能力全方位地获取知识，在多维化的信息空间中认识问题，而现有的信息处理工具(尤其是计算机)只具有在数字化的单维信息空间中处理问题的能力，这就产生了人类认识问题的认识空间与所用的处理问题的方法空间不一致的矛盾，产生了人类较难直接理解信息处理工具的处理结果，更难以把自己已有的感知经验或认知经验与处理工具的处理结果发生直接联系。因此人们迫切地希望突破现有数字计算机只能处理单维的、数字化信息的限制，而把它扩展成具有处理多维信息的能力。换言之，在未来的信息社会里，人类希望自己在一个以人为中心的多维信息空间中去处理问题和提高认识。人的感知系统、认知系统、人类以往的经验与知识以及虚拟现实系统都是构成多维信息空间的组成部分。

为了说明多维信息空间的构成，不妨把它与传统的单维信息空间作一比较。在数字化的单维信息空间内，信息处理工具(或环境)是计算机，人与计算机是通过键盘、二维鼠标和显示屏幕等发生联系的，人类以往的经验是以数字化形式存储在数据库内的。在适人化

的多维信息空间内，信息处理工具(或环境)是虚拟现实系统，人与虚拟现实系统是通过各种先进的传感器和作用器发生联系的，人类以往的经验与体验(全部的感性知识和理性知识—)都是理解问题、寻求答案和提出新概念的基础。

一个理想化的多维信息空间的构成模型及其与虚拟现实系统的关系如图 18.1 所示。

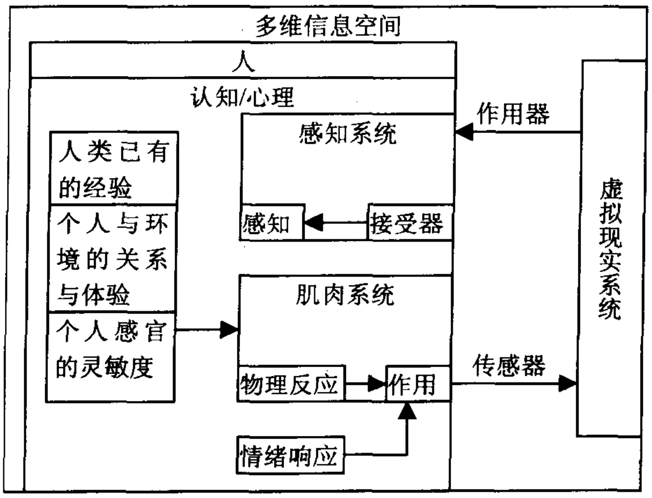

图18.1 多维信息空间的构成模型

其中人通过传感器把自己的经验和体验传送给虚拟现实系统，虚拟现实系统通过作用器把处理结果输出给人。人基于过去已有的对该物理情景的经验、人在该物理环境中的现时体验以及虚拟现实系统的现时输出，在虚拟现实系统的帮助下，经过综合集成获得了对该客观世界的认识和提高，虚拟现实系统对处理这类问题的能力也得到同步增长。

客观而论，目前虚拟现实技术所取得的成就，绝大部分还仅限于扩展了计算机的接口能力，还刚开始涉及到人的感知系统和肌肉系统与计算机的交互作用问题，还根本上未涉及“人在实践中得到的感觉信息是怎样在人的大脑中存储和加工处理，成为人对客观世界的认识的过程。只有真正涉及并开始找到对这些问题的技术实现途径时，人和信息处理系统之间的隔阂才能被彻底地克服。到那时，信息处理系统就不是一个只能处理数字化的计算装置，而将是一种具有对多维信息处理功能的机器，它将是人进行思维和创造的助手，它将是人对他们已有的概念进行深化和获取新概念的有力工具。

18.1.3 实现虚拟现实的软硬件平台

1. 硬件系统

- 跟踪系统

用于确定用户的头(头盔)、手(数据手套)、身体及任意真实世界物体的位置与方向。 目前主要包括磁性跟踪设备、声学跟踪设备、惯性跟踪设备、机械跟踪设备和光学跟踪设备等。

- 触觉系统

为虚拟世界提供手工探测或操作物体的现实感的生成设备，它利用人的触觉、力觉及

运动神经系统与虚拟环境相互作用，并具有如下 3 个功能：

测量用户手与/或其他部位的位置和受力，以便得到控制虚拟环境的输入。

显示受力和位置与/或它们对用户的空间和时间的分布。

通过力学作用单元产生力矩和力觉作用。

- 图形图像显示设备

虚拟环境应用中图像生成器的主要问题是延迟时间，即可视数据库或可视参数变化与屏幕显示变化间的延迟与实时显示要求之间的矛盾。影响延迟的 2 个主要因素是可视参数变化的计算开销和将数据送给显示装置的传送开销。传送速度的决定因素一是数据带宽和主频速度，二是从数据库中读取图形图像数据的速度。

- 可视化显示设备

主要包括 HMD 和其他立体显示设备。HMD 目前采用彩色液晶显示，但分辨率较低。

- 音频系统

主要功能包括制作、录音、再生和加工。在声音合成和定位方面仍需要进一步研究。

2. 软件系统

软件系统必须维护一个虚拟世界的实时模型，该模型能模拟虚拟物体本身的运动和不同虚拟物体间的交互作用关系，系统同时还必须提供实时计算能力。虚拟环境软件研究和实现中比较受重视的问题如下所示。

- 软件特性

许多商品化的软件系统可用于构造一些简单而专用的虚拟环境，即提供一个虚拟环境外壳。这个虚拟环境外壳应具有处理外部输入的能力、绘制能力、几何模型数据调度能力和物理变化模拟能力等。

- 工具

可以有许多成熟的模型和工具软件以及数据可以利用，另外利用网络上容易得到的免费软件往往也能提高效率。

- 设计和构造虚拟世界

构造虚拟世界的主要工作包括模型化和描述 2 大部分。描述生成主要使用 CAD 工具交互式构造，模型化是对世界动态性和交互性的抽象。

- 寻找合适的应用对象

在虚拟环境研究开发项目中，成功的经验是根据应用对象的具体情况研究开发针对性技术，然后在此基础之上抽象出一般方法。在虚拟环境投资方面，军事模拟远远超过了民用。迄今为止，实时军事模拟器系统都遵从共同的网络协议，称为分布式交互仿真。

18.1.4 虚拟环境人机交互系统

1. 设计准则

- 系统应该是开放的。
- 系统应该是面向对象的和可重用的。

- 系统应该是标准化的。
- 所选用的部件或子系统应该是有生命力的。
- 用户的安全和健康应放在第一位。

2. **人机接口系统的构成**

在理想情况下，虚拟环境系统应当是多传感器构成的人机交互方便的综合集成系统，该系统所产生的图像分辨率应该足够高，图像要丰富和自然，系统集成不会产生延迟，色彩和立体感好，音响的立体声效果好，语音合成逼真，压力细微，用户使用系统不会轻易感觉疲劳，系统中的传感器系统应具有多个自由度，延迟时间要足够短。一个理想的虚拟环境系统的人机交互系统应该包括如图 18.2 所示的部分。

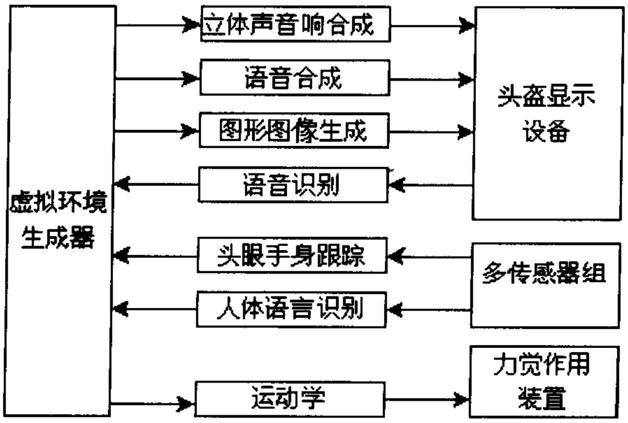

图18.2 虚拟环境系统的人机交互系统构成

18.2 虚拟漫游技术简介

当现实世界的影像、声音等信息都被数字化之后，变成了计算机中 0 和 1 的组合时，“虚拟现实”诞生了。因特网上风靡一时的虚拟旅游，正以各种方式，再造一个逼真的“另类现实”，大大解放了人们的想象力，同时也带给人们几许困惑。当我们安坐家中，单击手中的小小鼠标键，上网虚拟旅游，“身临其境”地游览美国的名胜景点时，会是怎样一番感受呢？

18.2.1 虚拟漫游系统示例

英特尔在其主页为网民们提供了一系列的虚拟漫游旅程，如中国的故宫、英国的远古石阵以及美国名胜、香港全景漫游等。以虚拟美国全景像旅程为例，有 2 种观赏方式，一种可提供 360° 全方位视角，景点有国会大厦、金门大桥、旧金山中国城等 5 处；2 种被称作“活照片”，可欣赏到奇丽的大峡谷和大名鼎鼎的好莱坞剧院。单击“国会大厦外观”的字样，开始网上虚拟美国游，只见电脑屏幕上出现了一张模糊的国会大厦的外观图，随着图片右下角一个白色地球仪状的物体的闪烁，图片逐渐变得清晰起来。过了几分钟时间，

这张显示内容量高达几百千兆的特制图片终于下载完毕，其清晰度果然不同凡响。在图片中央位置移动光标，可看见加减符号，轻轻单击，便进退自如“走”得近了，可看见原来只是一个黑点的。国会大厦阶梯上站着的一男一女游客；仰起头来，发现计算机屏幕可以整个地拥抱国会大厦的漂亮屋顶；徐徐背过“身”去，可瞧见横亘着的马路两边的情景，还有马路这边的行人、草坪和天空……给人的感觉是整个钻进了这一张全景图里、钻进了计算机屏幕里，真有点不可思议。

至于“活照片”一类的，观看起来要方便得多。照片动而人不动。人的视线，被放置在一个飞车走壁的特技高手的身上。随着黑色桶壁的渐渐退缩，大峡谷的蓝天白云、绿树彩岩，以及澄澈的湖光水色，不断地旋转着映入眼帘，使用户仿佛置身于这神奇美丽的大自然中，流连忘返……

威尼斯的虚拟三维模型，虽然没有照片效果的清晰度，但交互性方面更胜一筹。声称能让人乘着小船，体验泛波运河、穿过小桥的情景，感受动态的灯光和水纹的荡漾。

客观地讲，今日网上虚拟旅游的感觉，距离人们想象中宏伟而逼真的建筑群，从电脑屏幕上扑面而来，人像超人一般在虚拟空间里飞翔，自由得可以靠近每一扇窗户的梦想，还有很相当的距离。然而，无论是清晰如照片的虚拟全景像旅程，或者如同电脑游戏的三维场景模型，信息时代高科技的数码仿真技术，相信已能让人一瞥“虚拟现实”的绮丽风采，享受短暂的时空倒流的乐趣。

18.2.2　建筑物虚拟漫游

虚拟建筑物漫游是一种前景非常好的技术领域，例如，房地产开发商可以通过在网上向客户提供建筑物虚拟漫游招徕客户，同时也可以在房屋真正建议之前，给客户一个更加直观的感受，对房屋的销售一定大有裨益。同时建筑物虚拟漫游也为建筑师们提供了很大便利，他们可以通过对设计好的建筑进行反复观察，找出设计的不足之处，加以改进，以负在设计完成，交付施工后才发现有设计的缺陷而悔之晚矣。各种展览也可以非常简单地在虚拟漫游的环境下进行了，有了虚拟漫游的环境，人们可以在其中任意安排各种物品，达到各种效果。

最重要的，建筑物虚拟漫游使人们可以足不出户就能欣赏到世界各地的名胜古迹，也就是说建筑物虚拟漫游为普通人的娱乐增添了情趣。

目前，从技术角度来说，建筑物虚拟漫游的最大难点在于建模和实时绘制。由于建筑物漫游所看到的景象离人们很近，因此需要绘制得非常逼真，因此建模时需要构造得非常精细，这往往需要耗费很多时间。同样，对于构造出来的如此复杂的模型，在对其进行绘制时，由于机器性能的制约通常很难达到实时效果，这也是人们很难接受的。一般说来，人们需要在模型的精细程度和绘制的速度方面取一个折衷，既要保证一定的绘制质量，又不造成用户的运动不适感。另外还可以运用一些图形学的解决办法，例如考虑到建筑物漫游通常是室内环境，可以事先用辐射度方法对场景进行预处理，这样在漫游时就可以省去许多计算光照的计算量。再者，还可以运用 LoD(多层次细节)的方法为场景生成不同的细节层次，以便在不同的显示参数下调用不同的细节层次进行绘制，从而大大减小绘制的计算量。

本章所用的方法就是先用辐射度方法对场景进行预处理，在漫游时只要根据事先计算出来的能量值进行绘制即可。关于辐射度的计算由于过于繁琐，这里不作解释，有兴趣的用户可以参阅有关文献或书籍。这里只介绍读取辐射度计算的结果进行绘制和漫游的程序。用户可以参考这种方法，先编写辐射度程序对场景进行运算，将中间结果保存，再编写类似的场景漫游程序，实现建筑物漫游。这种方法非常适合硬件环境较差，而又想实现较为逼真的虚拟现实场景的情况。这种方法，至少可以让用户体验虚拟现实的诱人魅力，吸引用去做更有意义的探索。希望本章内容能够起到一个抛砖引玉的功能作用，希望更多的人对虚拟现实感兴趣。

18.3 虚拟建筑物漫游编程实例

通过上面的介绍可以知道，虚拟建筑物漫游是一个非常有趣的的应用方向，用户可以综合运用前面学到的一些知识，进行复杂应用程序的编写。在编程之前先对辐射度作简单的介绍，然后给出辐射度计算结果的存储格式，最后给出读取此数据进行漫游的程序。

18.3.1 辐射度方法简介

常见的真实感绘制方法有光线跟踪和辐射度等。光线跟踪的整体光照明模型成功地模拟了光在光滑景物表面之间的反射和折射过程。由于光线跟踪技术建立在空间点采样的基础之上，采样光线的数目受到很大限制，因此无法模拟光能在景物表面间由漫反射到漫反射的能量传播形式。这使许多照明效果得不到正确表现，较为明显的就是彩色渗透现象，即环境中景物的颜色相互作用使物体表面的颜色发生变化。

辐射度方法把组成环境的曲面作为一个封闭的系统，并假定封闭系统中的曲面均为漫反射面，然后根据能量平衡原理计算出每一曲面的能量，从而求出被观察点的光亮度。由于考虑了封闭系统中的各个曲面之间的能量传递，因而能够从整体上正确计算环境中的光能分布。

另外，由于辐射度方法在进行光亮度计算与画面绘制时所选择的观察位置和方向无关，因而辐射度方法是一种与视点无关的计算方法。这一特点用户可以在计算出某一场景的表面光亮度后，迅速地显示该场景不同观察角度的真实感视图。因此同光线跟踪技术的整体光照明模型相比，辐射度方法更有实用性。正是基于这个理由，本例的建筑物漫游程序选取了辐射度方法作为绘制方法，因为它可以在观察角度发生变化时迅速地生成场景的真实感视图。

18.3.2 数据存储结构

为了将辐射度生成的中间结果保存，让其后的绘制和漫游程序使用，设计了一种简单且高效的数据结构。

辐射度生成的结果以文件方式保存，下面给出了一种简单的文本描述方式。其中主要

描述了模型的几何造型，包括纹理文件、几何坐标、顶点坐标及其颜色和纹理坐标、面信息，其中辐射度的结果以颜色方式附加在每一个顶点上。

这样，就可以像其他的模型一样读取模型信息，再利用 OpenGL 进行绘制了，只是这时应该关闭光照，因为辐射度的结果无需再进行普通的光照计算。

以下是文件存储的具体结构，用近似伪代码的方法加以描述。

```
"texnum "  texnum  //  表明一个固定的字符串后跟一个该类型的整数字符串
TextureList[1]->fname  TextureList[1]->type//  纹理文件名和纹理类型
...                          //  重复以上内容
...
TextureList[texnum]->fname  TextureList[texnum]->type//  以上是一
             个列表
"objectnum "  objectnum  //  建筑物由多少个独立的物体构成
every object          //  这是一个循环定义，循环次数由 objectnum 决定
{
"object"  objectindex
"surfacenum "  surfacenum
every surface     //  这是一个循环定义，循环的次数由 surfacenum 决定
{
"surface "  surfceindex  //  面的编号
"textureid "  textureid  //  纹理的 ID
"pointnum "  pointnum  //  顶点数目
"triangle "  trianglenum  //  三角形数目
"quadrangle  "  quadranglenum  //  四边形数目
every point         //  循环定义，次数由 pointnum 决定
{
r g b u v x y z      //  每个顶点定义它的颜色值和纹理及几何坐标
...
}
every face       //  循环定义，由 trianglenum 和 quadranglenum 共同决定
{
v1 v2 v3         //  先定义 triangle
...          //  共 trianglenum 个
v1 v2 v3 v4       //  再定义 quadrangle
...          //  共 quadranglenum 个
}
}
}
```

18.3.3 读取辐射度数据

上面给出了辐射度数据的存储结构，可以看出结构非常清晰，据此可以很容易地编写读取此数据的程序，下面就介绍这段读取程序。由于涉及到很多从文件中进行字符串转换成数字的的操作，程序使用了 C 语言中最简单的 fscanf 函数。从文件中读取辐射度数据的程序代码如下：

```
void  CRenderView::ReadData()
{
    int     i,j,l;
    FILE    *fp;
    char    stemp[100];
    POINT3D   *plist;
    INT4U   nAllVertexNum;
    INT4U   *pchlist;
    strcpy(gEnergyFile,"room.ed");
    fp = fopen( gEnergyFile, "r" );
    if ( fp == NULL )
    {
    printf( "\n Can not open energy data file:%s\n", gEnergyFile);
         exit(0);
    }
    fseek( fp, 0, SEEK_SET);
    //  读取纹理列表
    fscanf( fp, "%s", stemp);
    while( strcmp( stemp,"texnum" ) != 0)  fscanf( fp, "%s", stemp);
    fscanf( fp, "%d", &texnum );
    TextureList  =  (TEXTURE_2D **)malloc  ( sizeof  (TEXTURE_2D)  *
            (texnum+1));
    for(i=1; i<=texnum; i++)
    {
TextureList[i] = (TEXTURE_2D *)malloc( sizeof(TEXTURE_2D));
fscanf( fp, "%s%s", TextureList[i].>fname, stemp );
if ( strcmp( stemp,"REPEAT_TEXTURE" ) == 0)
     TextureList[i]->type = 1;
else  if ( strcmp( stemp,"CLAMP_TEXTURE" ) == 0)
         TextureList[i]->type = 0;
    }
    //  读取独立物体列表
```

```
    fscanf( fp, "%s", stemp);
    while( strcmp( stemp,"ObjectNum" ) != 0) fscanf(fp,"%s",stemp);
    fscanf( fp, "%ld", &ObjectNum);
    ObjectList = (OBJECT *)malloc( sizeof(OBJECT ) * ObjectNum);
    for(i = 0; i < ObjectNum; i ++ )
    {
    fscanf( fp, "%s", stemp);
    while( strcmp( stemp,"SurfaceNum" ) != 0) fscanf(fp,"%s",stemp);
    fscanf( fp, "%ld", &(ObjectList[i].SurfNum) );
ObjectList[i].surflist =
   (SURFACE *)malloc( sizeof(SURFACE) * ObjectList[i].SurfNum);
for(j = 0; j < ObjectList[i].SurfNum; j ++ )  //  循环读取
{
//  读取块信息
            fscanf( fp, "%s", stemp);
    while( strcmp( stemp,"TextureId" ) != 0) fscanf(fp,"%s",stemp);
    fscanf( fp, "%d", &(ObjectList[i].surflist[j].texId) );
            fscanf( fp, "%s", stemp);
    while( strcmp( stemp,"pointnum" ) != 0) fscanf(fp,"%s",stemp);
    fscanf( fp, "%d", &(ObjectList[i].surflist[j].pointn) );
            fscanf( fp, "%s", stemp);
    while( strcmp( stemp,"triangle" ) != 0) fscanf(fp,"%s",stemp);
    fscanf( fp, "%d", &(ObjectList[i].surflist[j].triangle) );
            fscanf( fp, "%s", stemp);
    while( strcmp( stemp,"quadrangle" ) != 0) fscanf(fp,"%s",stemp);
    fscanf( fp, "%d", &(ObjectList[i].surflist[j].quadric) );
//  读取顶点信息
    ObjectList[i].surflist[j].pointlist = (POINT3D*)malloc(sizeof
            (POINT3D) *
ObjectList[i].surflist[j].pointn);
plist = ObjectList[i].surflist[j].pointlist;
for( l = 0; l < ObjectList[i].surflist[j].pointn ; l ++ )
fscanf(  fp,  "%f%f%f%f%f%f%f%f",  &(plist[l].r),  &(plist[l].g),
            &(plist[l].b),&(plist[l].u), &(plist[l].v),
&(plist[l].x), &(plist[l].y), &(plist[l].z) );
//  读取块列表
nAllVertexNum = ObjectList[i].surflist[j].triangle * 3 +
bjectList[i].surflist[j].quadric *4 ;
ObjectList[i].surflist[j].patchlist =
```

```
(INT4U *)malloc( sizeof(INT4U) * nAllVertexNum);
pchlist =  ObjectList[i].surflist[j].patchlist;
for( l = 0; l <  nAllVertexNum; l ++ )
fscanf( fp, "%ld", &(pchlist[l]) );
}
}
 fclose(fp);
}
```

这段程序代码并不复杂而且有注释，比较容易理解，在此不再赘述。

18.3.4 纹理映射相关程序

以下方法可以实现从若干个位图(.bmp)文件中读取纹理的功能。作为纹理的位图文件必须是 $2^n \times 2^n$ 大小。

```
extern TEXTURE_2D **TextureList;
//  按纹理索引值和纹理大小打开纹理
unsigned char *CRenderView::OpenTexImage( INT2U TexIndex,
            INT2U *rslx, INT2U *rsly )
{
unsigned char *image;
FILE *fp;
INT2U srcx, srcy;
INT4U i, j;
char ImageName[30];
unsigned char *SImageData;
int rc;
int width, height;
strcpy( ImageName, TextureList[TexIndex]->fname);
/* 装入位图 */
   fp = fopen(ImageName,"rb");
   if(!fp) return 0;
   fseek(fp,18L,0);
   rc=fread(&width,sizeof(long),1,fp);
   rc=fread(&height,sizeof(long),1,fp);
           *rslx=srcx=width; *rsly=srcy=height;
           fseek(fp,54L,0);
   image = (unsigned char *)malloc(width*height*3);
   rc=fread(image,width*height*3,1,fp);
   fclose(fp);
```

```
SImageData = (unsigned char *)malloc(srcx*srcy*3);
    for(i=0; i<srcx; i++) {
       for(j=0; j<srcy; j++) {
       (unsigned char)*(SImageData+i*srcx*3+j*3+0) =
       (unsigned char)*(image+i*srcx*3+j*3+2);
       (unsigned char)*(SImageData+i*srcx*3+j*3+1) =
       (unsigned char)*(image+i*srcx*3+j*3+1);
       (unsigned char)*(SImageData+i*srcx*3+j*3+2) =
       (unsigned char)*(image+i*srcx*3+j*3+0);
   }
    }
    free(image);
          printf("%s : %ld=%ld\n", ImageName, srcx*srcy*3,i*j*3);
          return( SImageData );
}

// 纹理初始化
void CRenderView::InitTex( int TexIndex )
{
INT2U  TextType;
unsigned char *ImageData;
static int OldIndex = -1;
if(TexIndex<=0) return;
if(TexIndex == OldIndex)
{
glEnable(GL_TEXTURE_2D);
return;
}
ImageData = ImageDatas[TexIndex-1];
TextType = TextureList[TexIndex]->type;
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
if( TextType == CLAMP_TEXTURE )
{
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
```

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
}
else
{
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
}
glTexImage2D(GL_TEXTURE_2D,0,3,rslxs[TexIndex-1],rslys
            [TexIndex-1],
0,GL_RGB, GL_UNSIGNED_BYTE, ImageData );
glEnable(GL_TEXTURE_2D);
OldIndex = TexIndex;
}

// 释放纹理列表
void CRenderView::freelist()
{
int i, j;
for( i=0; i<ObjectNum; i++)
{
for( j=0; j<ObjectList[i].SurfNum; j++)
{
free(ObjectList[i].surflist[j].pointlist);
free(ObjectList[i].surflist[j].patchlist);
}
free( ObjectList[i].surflist );
}
free(ObjectList);
for(i=1; i<=texnum; i++)
free(TextureList[i]);
free(TextureList);
}

// 关闭纹理
void CRenderView::CloseTex()
{
glDisable(GL_TEXTURE_2D);
}
// 装入所有纹理
```

```
void CRenderView::LoadAllTexture()
{
   int i;
for (i=0; i <texnum ; i++)
 ImageDatas[i] = OpenTexImage( i+1, &rslxs[i], &rslys[i] );
}

// 释放所有纹理
void CRenderView::FreeAllTexture()
{
   int i;
for (i=0; i <texnum ; i++)
  free(ImageDatas[i]);
}
```

18.3.5 绘制程序

有了以上的读取数据的程序和纹理读取和定义程序之后，就可以编写图形绘制程序了。由于在辐射度的存储结果中已经保存了每个顶点的颜色值，不需要向前面的方法一样用加光照的方法去计算顶点颜色，只要利用 OpenGL 定义面和顶点的方法相应地进行几何信息定义即可，需要注意的是，这时一定要将光照关闭，同时定义顶点的几何信息之前先指定顶点的颜色。这段程序代码如下：

```
void CRenderView::Render(void)
{
  int  i, j, k, l, m, TexIndex;
  POINT3D   *plist;
  INT4U     *pchlist;
  glClear(GL_COLOR_BUFFER_BIT|
       GL_DEPTH_BUFFER_BIT|
       GL_ACCUM_BUFFER_BIT);
  for(i = 0; i < ObjectNum; i ++ )
for(j = 0; j < ObjectList[i].SurfNum; j ++ )
{
TexIndex = ObjectList[i].surflist[j].texId;
if( TexIndex > 0 )
InitTex( TexIndex );
plist = ObjectList[i].surflist[j].pointlist;
pchlist =  ObjectList[i].surflist[j].patchlist;
l = 0;
```

```
for ( k = 0; k < ObjectList[i].surflist[j].triangle; k ++)
{
glBegin( GL_TRIANGLES );
for( m = 0; m < 3; m ++ )
{
    glColor3f ( plist[pchlist[l]].r,
             plist[pchlist[l]].g,
plist[pchlist[l]].b );
   glTexCoord2f( plist[pchlist[l]].u,
   plist[pchlist[l]].v );
    glVertex3f( plist[pchlist[l]].x,
plist[pchlist[l]].y,
plist[pchlist[l]].z );
   l ++;
}/* m */
glEnd();
}/* k */
for ( k = 0; k < ObjectList[i].surflist[j].quadric; k ++)
{
glBegin( GL_QUADS );
for( m = 0; m < 4; m ++ )
{
   glColor3f ( plist[pchlist[l]].r,
   plist[pchlist[l]].g,
   plist[pchlist[l]].b );
   glTexCoord2f( plist[pchlist[l]].u,
   plist[pchlist[l]].v );
   glVertex3f( plist[pchlist[l]].x,
   plist[pchlist[l]].y,
   plist[pchlist[l]].z );
  l ++;
}/* m */
glEnd();
}/* k */
glFlush();
CloseTex();
}
}
```

18.3.6 漫游动作控制

作为一个漫游程序，当然应该能够根据用户的各种交互操作作出相应的场景漫游变换，对于普通配置的微机，程序主要响应的交互工具主要是键盘和鼠标。由于鼠标操作更为方便，这里只介绍鼠标的控制。

对场景显示的漫游控制在 OpenGL 中可以通过设置视点及其观察方向来确定。具体的，可以利用 gluLookat()函数，通过设置相应的参数来达到漫游的效果。程序开始从文件中读入视点信息的初始设置值，视点参数文件中的各项依次为：仰角、近平面、视点三维坐标和焦点三维坐标。读取并设置初始视点参数的程序代码如下：

```
void CRenderView::InitLookAt()
{
  FILE      *fp;
  strcpy(sLookAtFN,"room.lk");
  fp = fopen(sLookAtFN, "rb");
  if (fp == NULL)
  {
ex = ey = ez =1.0f;
cx = cy = cz =0.0f;
Near = 0.1f;
angle = 30.0f;
  }
  else      fscanf(fp, "%f%f%f%f%f%f%f%f",
           &angle, &Near, &ex, &ey, &ez, &cx, &cy, &cz);
  fclose(fp);
}
```

接着，在程序中规定了鼠标左键按下时移动鼠标时，向上移动表示前进，向下表示后退；鼠标右键移动时，向左、右移动分别表示逆时针或顺时针旋转。每处理完一次鼠标移动事件后记载下当前鼠标位置，以便后面判断相对运动，这一点在编写漫游类的程序时非常重要。

```
void CRenderView::OnMouseMove(UINT nFlags, CPoint point)
{
  switch(nFlags){
    case(MK_LBUTTON):
MoveEye(FORWARD,(GLfloat)(oldmy.point.y)/5.0f,1);
break;
    case(MK_RBUTTON):
      MoveEye(TURNLEFT, (GLfloat)(oldmx.point.x), 1);
      break;
```

```
    }
    oldmy = point.y;
    oldmx = point.x;
    Invalidate(FALSE);
    CView::OnMouseMove(nFlags, point);
}
```

以下这段程序代码就是根据视点变换类型的不同，相应地修改 gluLookat 参数来最终实现漫游功能。具体的，所谓向前移动就是视点沿视点当前位置朝焦点所在位置移动一定距离，然后焦点再移动相等距离；旋转操作事实上就是焦点绕视点作一次旋转操作，旋转的量由鼠标移动的变化量决定；俯仰变换与旋转变换类似。程序代码如下：

```
// Move Ege 函数移动眼睛或改变视角如果 update≠0
void CRenderView::MoveEye(int type, GLfloat amount, int update)
{
  GLfloat r, deltax, deltaz, angle;
  switch(type){
    case FORWARD:
            r = sqrt((cx-ex)*(cx-ex)+(cz.ez)*(cz-ez));
        deltax = amount*(cx-ex)/r;
        deltaz = amount*(cz-ez)/r;
            ex += deltax;
            ez += deltaz;
            cx += deltax;
            cz += deltaz;
      break;
    case TURNLEFT:
              angle = amount/360.0f;
        cx = (cx-ex)*cos(angle)+(cz-ez)*sin(angle)+ex;
        cz = (cz-ez)*cos(angle)-(cx-ex)*sin(angle)+ez;
      break;
    case UP:
      ey += amount;
      break;
    case LOOKUP:
      cy += amount;
      break;
  }
  if (update){
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
```

```
            gluLookAt(ex, ey, ez,
                      ex+cx, ey+cy, ez+cz,
                      0.0f,1.0f,0.0f);
        }
    }
```

程序的其他部分内容与前面介绍的其他程序大同小异，在此也不作过多讲解。最后给出程序生成的漂亮结果。

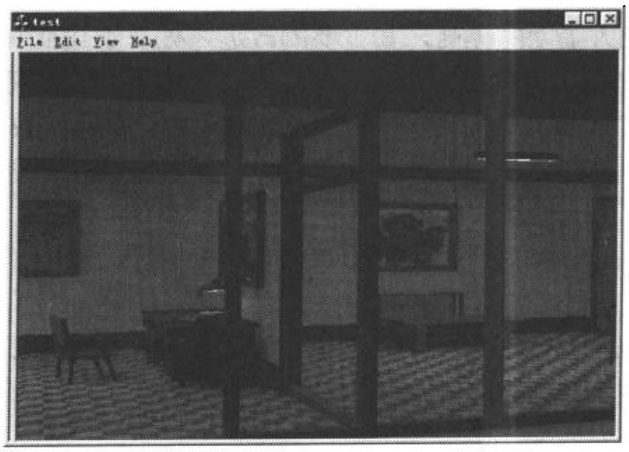

图18.1　虚拟建筑物漫游场景

第 19 章　OpenGL 新版本的新增功能

OpenGL 的版本现在已有 1.0，1.1 和 1.2 三个不同的版本，与 OpenGL 1.0 版相比，后 2 个版本的功能有了较大幅度的提高，其主要的改进之处在于功能有所增强，速度得到优化和提高。OpenGL 1.1 版本和 1.2 版本(由于 1.2 和 1.1 的差别不是特别大，以下将之统称为新版本)提供了许多 OpenGL 1.0 所没有的新增函数，目前这些函数在其他资料中介绍 TJ 很少。本书作者结合有关资料和应用体会对 OpenGL 新版本的新增函数进行总结和介绍，并以示例配合说明。

本节重点介绍 OpenGL 新版本新增的以下内容：

- 纹理处理功能

这些功能可以用 OpenGL 1.0 的函数组合扩充实现，在 OpenGL 新版本中用单独的函数形式加入到了 OpenGL 函数库中。

- 多边形偏移 (Polygon Offset) 功能

这些功能用以处理多边形和其边界线关系，以获得清晰的多边形形状。

- OpenGL 新版本的顶点数组

顶点数组处理功能用于高效率处理顶点数组，从而使得 OpenGL 的绘图效率更高。

19.1　新增纹理处理功能

OpenGL 新版本提供了新的纹理处理方法，这些方法在本节中分 7 部分介绍：

- 新增纹理格式。
- 使用纹理环境的新方法。
- 纹理代理 (Texture Proxy) 机制。
- 使用纹理物体 (Texture Object)。
- 使用子纹理 (Subtexture)。
- 从帧缓冲区(Framebuffer)中装入纹理。
- 其他新的扩展函数。

19.1.1　新增纹理格式

新增纹理格式是用于提高纹理使用性能，使整个纹理场景绘制效率更高。在使用 OpenGL 1.0 时纹理的定义方式为调用函数：glTexImage1D()和 glTexImage2D()。这 2 个函数的纹理颜色分量参数的值只能为 1~4 的整数。1 表示选择了 R 分量，2 表示选择了 G 分量，3 表示选择了 B 分量，4 表示选择了 R、G、B、A 四个分量。在 OpenGL 新版本中提

供了一些新增的预定义的格式，其优越性主要体现在以下 2 个方面。

节约内存空间

使用新的格式可以最大限度地适应系统的运行，新的纹理格式使用的纹理内存更小，从而可以把更多的纹理装入纹理内存中。这在 OpenGL 的绘图过程中，相当于在一定的显示模式下选择了最理想的纹理存储方式，即 8 位显示模式使用 8 位的纹理存储，16 位显示模式使用 16 位的纹理存储增加运算速度。OpenGL 新版本提供了一种十分细致的纹理大小的选择。在实际应用过程中，有时图形场景的绘制速度比图像质量显得更重要。在这种情况下，16 位的纹理比 32 位的纹理更符合实际，因此，在图像质量足够高的前提下，应该选择较小的纹理。OpenGL 新版本的这种纹理的区分增加了应用程序的运算速度。使用新增纹理格式的具体方法是在调用 glTexImage1D()或 glTexImage2D()时，在纹理颜色分量参数处使用新定义的参数宏。表 19.1 列出了 OpenGL 新版本新的宏名。

表19.1 OpenGL新版本新增的纹理内部参数格式

宏 名	基本格式
GL_LUMINANCE8	GL_LUMINANCE
GL_LUMINANCE4_ALPHA4	GL_LUMINANCE_ALPHA
GL_LUMINANCE12_ALPHA4	GL_LUMINANCE_ALPHA
GL_LUMINANCE16_ALPHA16	GL_LUMINANCE_ALPHA
GL_INTENSITY8	GL_INTENSITY
GL_RGB4	GL_RGB
GL_RGB8	GL_RGB
GL_RGBA4	GL_RGBA
GL_RGB5_A1	GL_RGBA
GL_RGBA8	GL_RGBA

表 19.1 列出了 OpenGL 新版本提供的新增宏名与其相应的基本格式，在 OpenGL 应用程序中可以使用任一宏来确定分量数，如果确定了分量数则宏的格式与表中的不完全相同。

19.1.2 使用纹理环境的新方法

在 OpenGL 1.0 中使用纹理环境的方法是通过调用函数 glTexEnv()来实现的。通过调用这一函数，可以对纹理进行调制(Modulating)和融合(Blending)。在 OpenGL 1.0 中使用调制和融合主要的作用为：

- 绘制表面。
- 调制表面非纹理的颜色。
- 将纹理映射中的颜色和非纹理颜色相混合。

OpenGL 1.0 中的纹理环境函数 glTexEnv()的第 3 个参数有 3 个宏：

GL_MODULATE，GL_DECAL 和 GL_BLEND。

这些宏的具体含义可以参见有关手册，本书不作详细讲述，在 OpenGL 新版本中增加了新的纹理环境参数，其宏名为：

GL_REPLACE，GL_ALPHA 和 GL_INTENSITY。

表 19.2 列出了在不同纹理环境下使用不同的纹理格式或颜色分量值时的纹理函数式。

表19.2 不同环境下的纹理格式

基本纹理格式	替 代	调 制	融 合	贴 花
GL_LUMINANCE	Rv = Lt Gv = Lt Bv = Lt Av = Af	Rv = Rf * Lt Gv = Gf * Lt Bv = Bf * Lt Av = Af	Rv = Rf * (1.Lt) + Rc* Lt Gv = Gf * (1.Lt) + Gc * Lt Bv = Bf * (1.Lt) + Bc * Lt Av = Af	未定义
GL_ALPHA	Rv = Rf Gv = Gf Bv = Gf Av = At	Rv = Rf Gv = Gf Bv = Gf Av = Af * At	Rv = Rf Gv = Gf Bv = Bf Av = Af * At	未定义
GL_INTENSITY	Rv = It Gv = It Bv = It Av = It	Rv = Rf * It Gv = Gf * It Bv = Bf * It Av = Af * It	Rv = Rf * (1.It) + Rc * It Gv = Gf * (1.It) + Gc * It Bv = Bf * (1.It) + Bc * It Av = Af * (1.It) + AcIt	未定义
GL_LUMINANCE _ALPHA	Rv = Lt Gv = Lt Bv = Lt Av = At	Rv = Rf * Lt Gv = Gf * Lt Bv = Bf * Lt Av = Af * At	Rv = Rf * (1.Lt) + Rc * Lt Gv = Gf * (1.Lt) + Gc * Lt Bv = Bf * (1.Lt) + Bc * Lt Av = Af * At	未定义
GL_RGB	Rv = Rt Gv = Gt Bv = Bt Av = Af	Rv = Rf * Rt Gv = Gf * Gt Bv = Bf * Bt Av = Af	Rv = Rf * (1.Rt) + Rc * Rt Gv= Gf * (1.Gt) + Gc * Gt Bv = Bf * (1.Bt) + Bc * Bt Av = Af	Rv = Rt Gv = Gt Bv = Bt Av = Af
GL_RBGA	Rv = Rt Gv = Gt Bv = Bt Av = At	Rv = Rf * Rt Gv = Gf * Gt Bv = Bf * Bt Av = Af * At	Rv = Rf * (1.Rt) + Rc * Rt Gv = Gf * (1.Gt) + Gc * Gt Bv = Bf * (1.Bt) + Bc * Bt Av = Af * At	Rv = Rf * (1.At) + Rt * At Gv = Gf *(1.At)+Gt*At Bv=Bf*(1.At)+Bt * At Av = Af

表 19.3 列出了纹理函数数式的缩写意义。

表19.3 纹理函数式缩写表示意义

缩 写				意 义
Rv	Gv	Bv	Av	纹理函数式运行结果
Rt	Gt	Bt	At	纹理颜色
Rf	Gf	Bf	Af	面片颜色
Rc	Gc	Bc	Ac	纹理环境颜色(GL_TEXTURE_ENV_COLOR)

19.1.3 纹理代理机制

在实际三维图形的应用开发时，需要用到大量的纹理贴图以营造真实的场景。OpenGL 1.0 中提供了使用纹理的方法。在实际应用中，会发现 OpenGL 1.0 的纹理方法效率仍然很低，其主要原因是在程序中很难判断纹理图像是否与纹理内存相符合。即使用参数为 GL_MAX_TEXTURE_SIZE 的函数 glGetIntegerv()，也不能得到精确的结果。因为，决定纹理是否与纹理内存相符合的因素包括纹理像素格式、颜色成分分量以及纹理的尺寸等。尤其是在 OpenGL 三维图形加速卡的支持下，这种现象尤为突出。OpenGL 新版本根据目前计算机三维图形的开发应用环境，即大多数计算机配备图形硬件加速卡的情况，充分利用硬件条件。为了达到这个目的，必须确定纹理最大限度地与纹理内存相符合。这就是纹理代理(Texture Proxy)机制方法。纹理代理机制的实质是确保任何像素格式、颜色成分分量的纹理与纹理内存相符合。纹理代理机制的具体使用方法和步骤为：

(1) 调用函数 glTexImage2D()或 glTexImage1D()定义纹理作为代理，在函数中使用 GL_PROXY_TEXTURE_2D 或 GL_PROXY_TEXTURE_1D 参数，如：

```
glTexImage2D(GL_PROXY_TEXTURE_2D, level, components, width, height,
             border GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```

(2) 调用函数 glGetTexLevelParameteriv() 或 glGetTexLevelParameterfv()以测试代理纹理的状态值，确定代理纹理是否与纹理内存相符合。代理纹理在纹理内存的存储形式：其中，目标纹理 target 参数为代理纹理 GL_PROXY_TEXTURE_2D 或 GL_PROXY_TEXTURE_1D，Pname 参数为所需要返回值的参数名，如需要纹理宽度为 GL_TEXTURE_WIDTH，如果返回值为一个非 0 值，则代理纹理与纹理内存相符合。如：

```
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D , level , GL_TEXTURE_
WIDTH, &proxyWidth);
```

如果纹理太大，所有的替代纹理的状态变量值为 0。如果该纹理被采用，返回值则作为所需要的参数。1D 纹理代理的用法与 2D 纹理代理的方法类似。对于 1D 纹理代理而言，pname 不包括 GL_TEXTURE_HEIGHT。纹理代理过程仅是一个状态测试及返回值求解的过程，在实际纹理贴图时，仍然必须调用以 GL_TEXTURE_2D 为目标参数的 glTexImage2D() 函数来定义实际纹理。

(3) 对 mipmap 纹理的情况，为获得 mipmap 纹理的最大数组的大小，可在纹理代理过程中指定和查询在最高层中纹理代理的值，从而保证纹理的方位比率。在例 19.1 中的程序代码段说明了纹理代理机制的用法，这段程序判断大纹理按照给定的内部图像格式和

方位比率存放。

例 19.1 纹理代理机制的使用

```
// 演示纹理代理的使用，以检测纹理空间的大小，从而判断可以定位的最大纹理
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <GL/gl.h>
#include <GL/glu.h>
static int intFloorLog2(unsigned int x){
  int a = 0;
  while (x >>= 1) ++a;
  return a;
}
// 如果 X 是 2 的整数次幂返回 true
static GLboolean isPow2(unsigned int x)
{
  return ((x > 0) && (x & (x . 1) == 0));
}
// 按指定的内部格式和比例找到最大纹理
static void findMaxTexture(GLenum internalFormat, int dim,
int xAspect, int yAspect, int border,
int maxMipmapLevel, int *widthOut, int *heightOut)
{
int level=0, levelOffset = 0;
int width=1, height = 1;
int maxLevel=0, maxWidth = 0, maxHeight = 0;
if (xAspect > yAspect) {
     width = xAspect / yAspect;
     levelOffset = intFloorLog2(width);
  } else {
     height = yAspect / xAspect;
     levelOffset = intFloorLog2(height);
  }
  while (1) {
    GLint proxyComponents;
    switch (dim) {
      case 1:
     glTexImage1D(GL_PROXY_TEXTURE_1D, level,  internalFormat,
           width+2*border,  border, GL_RGBA,  GL_UNSIGNED_BYTE,
```

```
            NULL);glGetTexLevelParameteriv(GL_PROXY_TEXTURE_1D,
      level, GL_TEXTURE_COMPONENTS,
       &proxyComponents);
      break;
      case 2:
      glTexImage2D(GL_PROXY_TEXTURE_2D,
      level , internalFormat , width+2*border , height+2*border ,
            border,
                   GL_RGBA, GL_UNSIGNED_BYTE,
                   NULL);
      glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, level,
      GL_TEXTURE_COMPONENTS, &proxyComponents);
         break;
         default:
          *widthout = 0;
          *heightout = 0;
           return;
       }
       if (proxyComponents != internalFormat) {
         /* 申请纹理代理失败，跳出循环 */
         break;
       } else {
         /* 申请纹理代理成功 */
         if (level>maxLevel || width>maxWidth || height>maxHeight)
         {
           maxLevel = level;
           maxWidth = width << level;
           maxHeight = height << level;
         }
       }
   if (maxMipmapLevel == 0) {
   /* 在第 0 层试验下一个更大的图像 */
   width <<= 1;
   height <<= 1;
   } else {
   /* try same image size at next higher mipmap level */
   ++level;
   if (level+levelOffset > maxMipmapLevel)
   /* 不能查询不存在的层，跳出循环 */
```

```
break;
}
}
}
if (maxWidth>0 && maxHeight>0) {
maxWidth += 2*border;
maxHeight += 2*border;
}
*widthOut = maxWidth;
*heightOut = maxHeight;
}
static void
displayTextureSizeInfo(void)
{
GLint maxTextureSize, maxTextureLevel;
int maxWidth, maxHeight;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxTextureSize);
printf("GL_MAX_TEXTURE_SIZE: %d\n", maxTextureSize);
maxTextureLevel = intFloorLog2(maxTextureSize);
findMaxTexture(GL_RGB, 2, 1, 1, 0, 0, &maxWidth, &maxHeight);
printf("RGB 1: 1 (%d x %d)\n", maxWidth, maxHeight);
findMaxTexture(GL_RGBA, 2, 2, 1, 0, 0, &maxWidth, &maxHeight);
printf("RGBA 2: 1 (%d x %d)\n", maxWidth, maxHeight);
findMaxTexture(GL_RGB, 2, 1, 1, 0, maxTextureLevel, &maxWidth,
&maxHeight);
printf("RGB 1: 1 (%d x %d) mipmap\n", maxWidth, maxHeight);
findMaxTexture(GL_LUMINANCE_ALPHA, 2, 1, 1, 1, 0, &maxWidth,
&maxHeight);
printf("LUMINANCE_ALPHA 1: 1 (%d x %d) w/border\n", maxWidth,
maxHeight);
}
```

19.1.4 使用纹理物体

在 OpenGL 新本版中对纹理性能的改进和提高使用了多种新的方法，纹理物体就是用于改进纹理运行性能的又一重要方法。纹理物体的功能主要在使用多种物体时得到充分体现，在实际应用时可以创建多个纹理并且相互之间进行切换。为使用纹理物体，OpenGL 新版本定义了一个优先级参数，用这个参数可以确定纹理在硬件纹理内存中进行分配和使用的优先级别。本节首先介绍纹理物体的工作机制，然后介绍纹理物体的使用方法。

1. 纹理物体的工作机制

在 OpenGL 1.0 中每个纹理目标 GL_TEXTURE_1D 和 GL_TEXTURE_2D 只能指定一个纹理。如果在绘制的场景中包含许多二维纹理，则需要对每个纹理通过调用函数 glTexImage2D()进行绘制。如果要使每个纹理具有不同的参数，如 WRAP 参数、过滤参数等，也需要对纹理进行重新定义。在 OpenGL 1.1 中 使用纹理物体的方法可以创建许多纹理，在创建过程中，对一个纹理物体定义一个名称(一个正数)就可以定义纹理及其参数。在绘制场景时，把纹理物体所对应的名称与纹理目标连接在一起即可。由于这种通过名称连接纹理的方法比定义纹理的方法更节省时间，这样很大地提高了纹理的使用效率，尤其是在需要纹理之间进行切换的时候。需要注意的是：纹理物体与显示列表属于不同的命名空间，不过，调用显示列表名称时，也可同时共享纹理物体。

2. 使用纹理物体

为创建纹理物体，需要用一个先前没有用过的名字命名纹理物体，这个命名将保留至纹理被删去为止。在下面的源代码中，用名称 1 和名称 2 定义纹理物体，并示范了纹理物体的定义和连接方法。

例 19. 2 使用纹理物体

```
/* 创建纹理物体 并命名为 1 */
glBindTexture( GL_TEXTURE_2D,  1 );
/* 定义纹理及其参数 */
glTexImage2D( GL_TEXTURE_2D,  0,  4,  32,  32,  0,  GL_RGBA,  GL_BYTE,
            img1 );
glTexParameteri( GL_TEXTURE_2D,  GL_TEXTURE_WRAP_S,  GL_CLAMP );
/* 创建纹理物体，并命名为 2 */
glBindTexture( GL_TEXTURE_2D,  2 );
/* 定义纹理及其参数 */
glTexImage2D( GL_TEXTURE_2D,  0,  4,  64,  64,  0,  GL_RGBA,  GL_BYTE,
            img2 );
glTexParameteri(  GL_TEXTURE_2D  ,   GL_TEXTURE_MAG_FILTER  ,
            GL_NEAREST );
/* 绘制场景时使用纹理物体 */
/* 调用纹理物体 1 */
glBindTexture( GL_TEXTURE_2D,  1 );
<绘制图元>
/*调用纹理物体 2 */
glBindTexture( GL_TEXTURE_2D,  2 );
<绘制图元>
```

3. 纹理物体的命名

纹理物体的命名原则是不与任何已经存在的纹理物体的名称相同。为判定当前名称是否已经使用，OpenGL1.1 提供了几个纹理物体管理函数。

判定一个给定的名称是否已经使用：

```
glIsTexture();
```

返回已经使用的纹理名称集：

```
glGenTextures();
```

删除一个或多个纹理物体：

```
glDeleteTextures();
```

调用该函数时删除了纹理物体，同时纹理物体的名称也释放。

4. 编辑和查询纹理物体

一旦创建了纹理物体，在应用程序中可以改变纹理物体的图像及其纹理参数。编辑纹理物体的方法与定义纹理物体的方法相同，即：

- 调用函数 glBindTexture()连接纹理物体。
- 调用 glTexImage*() 或 glTexParameter*()编辑纹理图像和纹理参数。

这个编辑过程对当前与纹理目标相连接的纹理物体起作用。在此同时，也可对纹理的参数和纹理图像内容进行查询：

- glGetTexParameter*()：获取纹理参数。
- glGetTexLevelParameter*()：获取纹理层参数。
- glGetTexImage()：获取纹理图像。

这些函数对当前已连接的纹理物体作用。为确定当前是什么纹理物体与纹理目标相连接，可调用函数 glGetIntegerv()，其参数为：

GL_TEXTURE_BINDING_1D 或 GL_TEXTURE_BINDING_2D。

5. 纹理优先级和驻留

在许多应用系统中，为达到系统运行速度的要求，在纹理使用之前常常把纹理驻留在硬件纹理内存中。在使用纹理物体连接纹理时，如果纹理内存不能存放所有纹理，系统需要把某些纹理先释放以腾出空间。这就是纹理的优先级问题。OpenGL 新版本提供了优先级函数用于指定纹理驻留于内存的优先级。这个函数是 glPrioritizeTextures()。

每个优先级值在 0~1 之间，0 表示最低级别，1 表示最高级别。设置纹理优先级的方法是通过调用带参数为 GL_TEXTURE_PRIORITY 的纹理参数设置函数 glTexParameter*() 来实现。同样，OpenGL 1.1 也提供了查询纹理物体是否驻留于内存的函数：

```
glAreTexturesResident();
```

这个函数用于查询单个纹理物体是否通过函数 glGetTexParameter*()驻留于内存。调用 glGetTexParameter*()时，其参数为 GL_TEXTURE_RESIDENT。

6. 默认纹理

使用纹理物体也可以得到像 OpenGL 1.0 一样的效果，即所谓的默认纹理，默认纹理就是把纹理物体命名为 0。在创建一个绘制描述表时，默认纹理可用于初始化连接；在创建一个纹理物体时，纹理物体的图像和参数与默认纹理的初始化设置相同。使用默认纹理时，使用函数 glTexParameter*()为默认纹理设置优先级。对于默认纹理，不能用函数 glPrioritizeTextures()设置。同样可以用函数 glGetTexParameter*()查询默认纹理驻留内存的状态，但不能用函数 glAreTexturesResident()。

19.1.5 使用子纹理

OpenGL 新版本提供了一种快速更新纹理图像的方法，用于一些特殊的应用，这种方法就是子纹理方法。子纹理方法可以在不影响纹理的其他参数的情况下，用一个新的图像替代原有图像的子矩形区域。在希望不改变纹理的其他参数，仅需要改变纹理的图像而重新定义纹理时，子纹理方法是十分有效的方法，这种方法比调用 glTexImage*()要快得多。子纹理方法在以下情况下可发挥特殊作用：

- 在处理纹理的图像与纹理内存不相匹配时，可以使用合适的纹理的一部分装入内存。
- 在纹理图像宽和高不是 2 的幂时，纹理图像不能直接作为纹理使用。这时可使用子纹理解决，通常的方法是建立一个 NULL 纹理，这个纹理的大小可直接作为纹理参数，然后把需要的纹理图像作为子纹理装入。
- 在处理动画时，装入大图像会影响速度，可在各帧内装入多个子纹理，这样动画的速度可以得到改善。本章中将介绍使用子纹理的方法和使用 NULL 图像作为子纹理的方法。

1. 使用子纹理

OpenGL 新版本提供了使用子纹理的函数，这些函数是：

```
glTexSubImage1D();
glTexSubImage2D();
glTexSubImage3D();
```

这些函数与以前纹理定义函数类似。这里以常用的二维子纹理函数进行说明。

```
void glTexSubImage2D(enum target, int level, int xoffset, int
            yoffset, sizei width,
      sizei height, enum format, enum type, const void* pixels);
```

其中有关参数说明如下：

- target 纹理目标参数，用于指定用作子纹理的图像。
- level，depth，format，type 和 pixels 与函数 glTexImage2D()的含义相同。
- width，height 子纹理的高度和宽度。
- xoffset，yoffset 指定需要调整的纹理图像在 X 和 Y 方向上的纹理像素的偏移量。

注意：在使用子纹理时，如果使用了纹理未初始化部分，可能会产生未知结果。

2. 与子纹理一起使用 NULL 图像

在实际应用中，有时需要在未初始化图像内容的情况下定义纹理图像的参数。例如，把 NTSC 制式的视频信号作为纹理。由于 NTSC 制式的视频图像的大小不是 2 的幂，因此首先定义一个比视频图像略大的区域，这个区域的高和宽符号纹理的要求，即为 2 的幂。在这种情况下，可以首先定义的区域图像为 NULL 图像。使用 NULL 图像的另外一种情况，需要先定义纹理的参数，再定义纹理的具体图像。对于这种情况，在定义纹理的参数后，使用 glTexSubImage*()函数定义具体的纹理图像。

创建 NULL 图像的方法与普通的纹理图像定义方法类似，即调用函数 glTexImage2D()等，不同的是函数中指向图像数据的指针 pixels 设置为 NULL 指针即可。关于 glTexImage2D()等函数的具体使用方法可参见第 19 章。这样，即建立了需要的纹理，但是实际上并未处理任何像素。

注意：如果使用一个未经初始化的纹理会产生未知结果。

下列程序代码段说明先创建一个 NULL 图像的纹理，然后装入子纹理。

```
{
/* 定义图像数组 */
static unsigned char image[32][32][4];
/* 建立一个 256 × 256 的 NULL 纹理 */
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB5, 256, 256, 0,
GL_RGBA, GL_UNSIGNED_BYTE, NULL);
/* 从 100，110 起装入 32 × 32 的子图像 */
glTexSubImage2D(GL_TEXTURE_2D, 0, 100, 110, 32, 32,
GL_RGBA, GL_UNSIGNED_BYTE, image);
}
```

19.1.6 从帧缓存中装入纹理

在 OpenGL 1.0 中，纹理的图像数据是从主机内存中装入。在 OpenGL 新版本中提供了一种新的纹理图像装入方法，即从帧缓存中装入图像的方法。从帧缓存中装入图像，可以代替纹理图像的一部分或者整个图像。从帧缓存中装入纹理图像，可以实现多通道绘制。在这种绘制过程中，可使用所有的 OpenGL 绘图命令来处理纹理，从而发挥更有特色的图形功能。例如，可以绘制一个场景，把这个场景的绘制结果拷贝至纹理内存，然后把纹理贴到三角形面片上。通过对三角形面片的操作，可以对场景变形，把场景打散，或者沿某个复杂物体弯曲等操作。

拷贝纹理图像功能使用 2 个函数：

```
glCopyTexImage1D();
glCopyTexImage2D();
```

这 2 个函数可从当前指定的颜色缓冲区拷贝图像数据。这样纹理图像就不是来自主机

内存。

直接从帧缓存中拷贝图像，可调用以下 4 个函数：

```
glCopyTexImage1D();
glCopyTexImage2D();
glCopyTexSubImage1D();
glCopyTexSubImage2D();
```

这 4 个函数的形式十分类似。下面是常用的二维纹理拷贝说明：

```
void glCopyTexImage2D ( GLenum target, GLint level,
                        GLenum internalformat, GLint x, GLint y,
                        GLsizei width, GLsizei height, GLint border )
```

其中的参数含义为：

target, level 等同于 glTexImage2D()中的参数含义，但 target 参数不接收纹理代理参数 GL_PROXY_TEXTURE_2D，internalformat 等同于函数 glTexImage2D()中的参数 components 参数的含义。但 glCopyTexImage2D()不接收值 1、2、3 和 4，只接收 OpenGL 新版本中定义的内部格式。这些格式在表 19.1 中已经列出。

width, heigh, border 定义了从颜色缓冲区中读取的图像的大小信息，这个图像的大小为：

宽，width+2*border_pixel;

高，height+2*border_pixel;

x，y 定义左下角的窗口坐标。

像素的处理是在调用函数 glCopyPixels()之后进行，但这种处理在最后的转换前进行，即在贴图至面片之前。这时，所有的像素成分值被限制在值域[0, 1]之间。如果需要调用 glTexImage*() 或 glTexSubImage*()，所有的像素才进行进一步的处理。像素的顺序是屏幕坐标 x 对应于水平方向的纹理坐标 i；屏幕坐标 y 对应于垂直方向的纹理坐标 j。

19.1.7 其他新的纹理扩展函数

OpenGL 新版本提供了一系列的新的纹理扩展函数。这些函数中，部分是新增的，部分是在原来基础上改进的。这些函数小结如下所示。

新增纹理处理函数：

```
glGenTextures();
glDeleteTextures();
glIsTexture();
glBindTexture();
glPrioritizeTextures();
glAreTexturesResident();
glTexSubImage1D();
glTexSubImage2D();
glTexSubImage2D();
```

```
glCopyTexImage1D();
glCopyTexImage2D();
glCopyTexSubImage1D();
glCopyTexSubImage2D();
glCopyTexSubImage3D();
```

功能扩展纹理处理函数：

```
glTexImage1D();
glTexImage2D();
glGetTexLevelParameteriv();
glTexEnvf();
glTexEnvi();
glTexEnvfv();
glTexEnviv();
glGetTexLevelParameterfv();
glGetParameterTex*();
```

19.2 多边形偏移功能

OpenGL 新版本提供了一种处理多边形的新方法，即多边形偏移方法，这种方法主要有以下功能：

- 边线消隐绘制。
- 带边界的实体绘制。
- 在表面使用贴花。

使用这种方法可以绘制更清晰的图像，例如，需要绘制物体表面的轮廓线时，线条并不按照多边形那样进行栅格化，这样在进行深度缓冲区的操作时会引起深度值的微小的不同，其结果是线条的某些像素显示，而某些像素不显示。实际上，这是由于三角形面和边线具有不同的深度值。系统并不在绘制之前表明那些内容在前或在后，多边形偏移方法可以很好地解决此类问题。

本节将分 2 部分介绍多边形方法：

- 使用多边形偏移功能。
- 多边形偏移编程示例。

19.2.1 使用多边形偏移功能

OpenGL 新版本中使用多边形偏移功能的方法十分简单，下面将其作一下归纳。

- 调用函数 glEnable() 其参数为 GL_POLYGON_OFFSET，启动多边形偏移功能。
- 调用函数 glPolygonOffset()，该函数作用是在深度缓冲区中偏移多边形像素，这样多边形的线和点不再交叉，在绘制时可清晰地绘制出多边形轮廓线。

- 调用函数 glDisable() 其参数为 GL_POLYGON_OFFSET，关闭多边形偏移功能。
- 使用 glPolygonOffset()函数，可以移动面片的深度值，这个深度值是在多边形的绘制过程中产生的。而多边形则已经加了一个偏移增量，偏移增量是与多边形的斜向深度的最大绝对值是相对应的，这个绝对值则是在窗口坐标系中使用和计算。这种偏移有以下作用：
- ◆ 使在同一平面上的直线、点和多边形在绘制时没有交叉。这样直线既可在多边形之前也可在多边形之后绘制。
- ◆ 如果对每个多边形都使用不同的偏移量，多面体在绘制时不会有交叉。

19.2.2 多边形偏移编程示例

本小节通过一个示例介绍多边形偏移的编程方法。多边形偏移的重点是函数 glPolygonOffset()。该程序绘制 3 种图像：

- 左图像是一个网格实体图像。
- 中间图像是一个没有多边形偏移的隐藏线消隐的图像。
- 右图像是一个使用多边形偏移的隐藏线消隐图像。

例 19.3 多边形偏移编程示例代码段

```
#include <GL/gl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MAXQUAD 6
typedef float Vertex[3];
typedef Vertex Quad[4];
Quad quads[MAXQUAD] = { /* 定义立方体集的 6 个面 */
0, 0, 0,  1, 0, 0,  1, 1, 0,  0, 1, 0,
0, 0, 1,  1, 0, 1,  1, 1, 1,  0, 1, 1,
0, 0, 0,  1, 0, 0,  1, 0, 1,  0, 0, 1,
0, 1, 0,  1, 1, 0,  1, 1, 1,  0, 1, 1,
0, 0, 0,  0, 0, 1,  0, 1, 1,  0, 1, 0,
1, 0, 0,  1, 0, 1,  1, 1, 1,  1, 1, 0
};
#define WIREFRAME 0
#define HIDDEN_LINE 1
static int dimension = 3;
static void init (void)
{
/* 设置视窗参数 */
glMatrixMode(GL_PROJECTION);
```

```
gluPerspective(20, 1, 0.1, 20);
glMatrixMode(GL_MODELVIEW);
glTranslatef(0, 0, -15);
/* 设置相关状态 */
glEnable(GL_DEPTH_TEST);
glPolygonOffset(1.0, 2);
}
/* 绘制 3 个立方体 */
static void
draw_scene(int mx, int my) {
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glTranslatef(-1.7, 0.0, 0.0);
cubes(mx, my, WIREFRAME);
glPopMatrix();
glPushMatrix();
cubes(mx, my, HIDDEN_LINE);
glPopMatrix();
glPushMatrix();
glTranslatef(1.7, 0.0, 0.0);
glEnable(GL_POLYGON_OFFSET_FILL);
cubes(mx, my, HIDDEN_LINE);
glDisable(GL_POLYGON_OFFSET_FILL);
glPopMatrix();
}
static void
cubes(int mx, int my, int mode) {
int x, y, z, i;
/* 立方体随鼠标移动 */
glRotatef(mx / 2.0, 0, 1, 0);
glRotatef(my / 2.0, 1, 0, 0);
/* 绘制消隐边线的多边形 */
glTranslatef(-0.5, -0.5, -0.5);
glScalef(1.0/dimension, 1.0/dimension, 1.0/dimension);
for (z = 0; z < dimension; z++) {
for (y = 0; y < dimension; y++) {
for (x = 0; x < dimension; x++) {
glPushMatrix();
glTranslatef(x, y, z);
```

```
glScalef(0.8, 0.8, 0.8);
for (i = 0; i < MAXQUAD; i++)
draw_hidden(quads[i], mode);
glPopMatrix();
}
}
}
}
static void
fill(Quad quad)
static void
fill(Quad quad) {
glBegin(GL_QUADS);
glVertex3fv(quad[0]);
glVertex3fv(quad[1]);
glVertex3fv(quad[2]);
glVertex3fv(quad[3]);
glEnd();
}
static void
outline(Quad quad) {
/* 绘制带轮廓线的多边形 */
glBegin(GL_LINE_LOOP);
glVertex3fv(quad[0]);
glVertex3fv(quad[1]);
glVertex3fv(quad[2]);
glVertex3fv(quad[3]);
glEnd();
}
static void
draw_hidden(Quad quad, int mode) {
/* 轮廓线为白色 填充色为黑色 */
glColor3f(1, 1, 1);
outline(quad);
if (mode == HIDDEN_LINE) {
glColor3f(0, 0, 0);
fill(quad);
}
}
```

```
/* 如果需要，可在此加入处理输入消息的程序代码 */
draw_scene(deltax, deltay);
```

19.3　OpenGL 新版本顶点数组

OpenGL 1.0 提供了用于建立场景实体的一系列绘制函数(也就是绘制命令)，如设置顶点坐标、设置当前颜色、设置法向量坐标和纹理坐标等。用这些函数可以直接利用坐标构造实体，从而具有极大的灵活性。但是在 OpenGL 1.0 中，需要多次重复地调用这些绘制函数。OpenGL 新版本提供了一种把实体的顶点数据放入内存中的方法。这种方法把顶点数据先存入数组，然后通过调用一个简单的函数调用数组中的数据块，再绘制多个几何实体。OpenGL 新版本中提供的方法最多可指定 6 个这样的数组，每个数组存储边界标志、纹理坐标、颜色索引、法向量和顶点坐标。本节介绍 OpenGL 新版本中的顶点数组的功能，主要分以下几部分介绍：

- 指定顶点数组。
- 变换独立的数组单元。
- 从不同的数组中连接数组单元。
- 通过建立顶点网格绘制实体。
- 使用隔行数组。
- OpenGL 新版本及顶点数组扩展功能。

19.3.1　指定顶点数组

OpenGL 新版本提供了 6 个用于指定顶点数组的函数。这些函数是：

```
glVertexPointer();
glNormalPointer();
glColorPointer();
glIndexPointer();
glTexCoordPointer();
glEdgeFlagPointer();
```

这些函数用于处理顶点的法向量、颜色、颜色索引、纹理坐标以及边界坐标。这里先介绍函数 glVertexPointer()：

```
glVertexPointer(int size, enum type, sizei stride, void *pointer);
```

指定顶点坐标数组的位置和数据格式，其中：

- size 指定每个顶点的坐标维，其值必须是 2、3 或 4。
- type 指定数组中每个坐标的数据类型，其值必须是 GL_SHORT, GL_INT, GL_FLOAT 或 GL_DOUBLE，分别对应于 OpenGL 的数据类型 short, int, float 和 double。
- strid 指定指针与连续的顶点之间的字节偏移量，如果该值为 0，则顶点数据在数

组中结合紧密。

- pointer 指定指向数组中第一个顶点的第一个坐标的指针。

glNormalPointer() 指定法向量数组的位置和数据格式。
glColorPointer() 指定颜色成分数组的位置和数据格式。
glIndexPointer() 指定颜色索引数组的位置和数据格式。
glTexCoordPointer() 指定纹理坐标数组的位置和数据格式。
glEdgeFlagPointer() 指定边界标志数组的位置和数据格式。

表 19.4 是 size 和 type 的值。

表19.4 顶点数组维数和数据类型

绘图函数	大 小	数据类型
glVertexPointer	2，3，4	短整型，整型，浮点型，双精度
glNormalPointer	3	字节，短整型，浮点型，双精度
glColorPointer	3，4	字节，无符号字节，短整型，无符号短整型，无符号整型，浮点型，双精度
glIndexPointer	1	无符号字节，短整型，整型，浮点型，双精度
glTexCoordPointer	1，2，3，4	短整型，整型，浮点型，双精度
glEdgeFlagPointer	1	布尔型

其中的数据类型对应表为：

表19.5 OpenGL中数据类型与C语言对应表

OpenGL 中的类型	对应于 C 语言的类型
GL_BYTE	字节
GL_SHORT	短整型
GL_INT	整型
GL_FLOAT	浮点型
GL_UNSIGNED_BYTE	无符号字节
GL_UNSIGNED_SHORT	无符号短整型
GL_UNSIGNED_INT	无符号整型

如果 size 的值不是表中所包含的值，则函数返回值为 GL_INVALID_VALUE。如果 stride 的值为 0，数组单元按顺序存放；否则，指向 i 和 i+1 的数组单元相差 stride 个基本单元。

对每个绘图函数而言，pointer 都是指定指向数组中第一个顶点的第一个坐标的指针。对于一个单独的顶点数组，可以调用状态函数设置数组打开和关闭状态。这些函数为：

```
void EnableClientState ( enum array ) ;
void DisableClientState ( enum array ) ;
```

其中的 array 按照表 19.6 中的标志来指定所要起作用的函数。

表19.6 顶点数组状态标志

所指定的数组函数	开启标志
glVertexPointer	GL_VERTEX_ARRAY
glNormalPointer	GL_NORMAL_ARRAY
glColorPointer	GL_COLOR_ARRAY
glIndexPointer	GL_INDEX_ARRAY
glTexCoordPointer	GL_TEXTURE_COORD_ARRAY
glEdgeFlagPointer	GL_EDGE_FLAG_ARRAY

19.3.2 变换独立的数组单元

在 OpenGL 新版本中进行顶点数组中，为对第 i 个单元进行变换操作，提供了一个新的函数：

```
void glArrayElement ( int i );
```

对于一个顶点数组而言，各种绘图操作都随着指向数组单元 i 的的指针而起作用。这些绘图操作可以是：

```
glVertex[size][type]v();
glEdgeFlag[size][type]v();
glTexCoord[size][type]v();
glColor[size][type]v();
glIndex[size][type]v();
glNormal[size][type]v();
```

其中，size 是[2，3，4]中的一个值；type 是[s，I，f，d]中的一个值，对应于短整型、整型、浮点型、双精度数据类型。上面的绘图操作对应于坐标顶点、边界标志、纹理坐标、颜色、索引和法向量。这些绘图操作的执行方式和顺序和 OpenGL 1.0 中的绘图方式和顺序一样，即状态操作在前，顶点定义在后，函数 glVertex[size][type]v()在最后。如果在执行 glBegin()和 glEnd()之间对顶点数组进行变化，这样会影响在同一个 glBegin()和 glEnd()之间的 glArrayElement()的调用。也就是说，在同一个 glBegin()和 glEnd()之间调用 glArrayElement()有一个次序问题。

19.3.3 连接不同数组的数组单元

使用顶点数组的一个重要的优越性是可以直接绘制一连串几何实体。OpenGL 新版本提供了专门的绘图函数：

```
void glDrawArrays(enum mode, int first, sizei count);
```

其中，mode 指定所建立的实体类型， mode 的宏名与 glBegin()所使用的宏名是相同

的。first 和 count 表示从 first 开始到 count 结束个数组单元。在 OpenGL 应用程序中调用函数 glDrawArrays()，实际上是通过一个数组序列，从 first 开始到 count 止，对每个单元调用函数 glArrayElement()。

注意：在调用了 glDrawArrays()之后，对顶点数组而言，当前边界标志、纹理坐标、颜色、颜色索引、法向量坐标都变得不确定状态。这些状态属性只对那些还没有调用 glDrawArrays()函数的未启动顶点数组起作用。

19.3.4 通过建立顶点网格绘制实体

OpenGL 新版本还提供了一种使用顶点数组的绘图函数。以下介绍的就是这种新的函数：

```
void DrawElements(enum mode, sizei count, enum type, void *indices );
```

使用这一函数的前提是顶点数组是用前面所列的函数建立的，在这种情况下，可调用该函数绘制图形实体。这个函数所绘制的实体是基于一个顶点序列，序列使用了 count 个单元，这些单元的数据类型索引为*indices 。该函数参数的详细解释如下：

mode 图形实体的类型，即 GL_POINTS， GL_LINE_STRIP 等，即 glBegin()函数所接受的宏名。

count 所要绘制的单元数。

type 是宏 GL_UNSIGNED_BYTE，GL_UNSIGNED_SHORT 或 GL_UNSIGNED_INT 中的一个值。

*indices 与 type 相对应的数据类型索引，即分别为 ubyte，ushort，uint。

函数 glDrawElements()的作用过程是对在索引 indices 中的数组单元集，调用 glArrayElement()，直至 count 个单元。

注意：在调用了 glDrawElements()之后，对顶点数组而言，当前的边界标志、纹理坐标、颜色、颜色索引和法向量坐标都变得不确定状态。这些状态属性只对那些还没有调用 glDrawElements()函数的未启动顶点数组起作用。

19.3.5 使用隔行数组

为了使已经使用了顶点数组的应用程序的运行速度更快，OpenGL 新版本中还提供了一个函数 glInterleavedArrays()。这个函数可以把几何体指定为 OpenGL 的一个独立的调用，从而使得速度更快。使用该函数不需要对每一类顶点数组进行单独的调用，另外，该函数的一个重要功能是把顶点数组直接包装起来，也称为隔行数组。使用这种包装的顶点数组的运行速度更快。

glInterleavedArrays()的函数形式为：

```
void glInterleavedArrays ( enum format , sizei stride , void
*pointer );
```

其中的参数含义为：

stride 如果该值为 0，单元集中单元之间没有空隙，否则单元之间存在 stride 个字节

的空隙。

*pointer 指向数组集的指针。

format 用于描述从数组集中抽取单个数组的参数，各参数如下所示：

```
V2F
V3F
C4UB_V2F
C4UB_V3F
C3F_V3F
N3F_V3F
C4F_N3F_V3F
T2F_V3F
T4F_V4F
T2F_C4UB_V3F
T2F_C3F_V3F
T2F_N3F_V3F
T2F_C4F_N3F_V3F
T4F_C4F_N3F_V4F
```

在 format 中指定这些参数的含义如表 19.7 所示。

表19.7 format中指定的参数的含义

format 中包含的字符	相应的含义
T	纹理坐标从隔行数组中抽取
C	抽取颜色值
N	抽取法向量值
V	抽取顶点坐标

参数中的数字 2，3，4 指定抽取多少个值。F 表示抽取的值为浮点数。如果 4UB 在 C 之后，则表示颜色数被抽取 4 个 unsigned 字节。如果一个颜色被抽取了 4 个 unsigned 字节，则随后的顶点数组位于浮点数所排列的地址的最前面。

19.3.6 顶点数组扩展功能

在 OpenGL 1.0 中，OpenGL 的扩展 EXT_vertex_array 在其所支持的系统中提供了顶点数组功能。本节说明顶点数组功能在 OpenGL 新版本以及扩展集中的区别。OpenGL 新版本提供了新的函数 glDrawElements()和 glInterleavedArrays()。这些函数在扩展集中没有相对应的函数。

glEnableClientState()和 glDisableClientState()也是 OpenGL 新版本中的新函数，在扩展集中则通过 glEnable()和 glDisable()来开启和关闭单独的数组单元实现。

OpenGL 新版本中有一些函数带 count 参数，这些函数是：

```
glEdgeFlagPointer();
glTexCoordPointer();
glColorPointer();
glIndexPointer();
glNormalPointer();
glVertexPointer();
```

count 参数表示数组中的单元数。这种功能在扩展集中则是以另外形式表达。

注意：这些 OpenGL 1.x 所支持的新的功能与扩展集是不能混用的，否则会产生不稳定的结果。

综上所述， OpenGL 新版本提供的与新函数如下所示：

```
glArrayElement();
glDrawArrays();
glVertexPointer();
glNormalPointer();
glColorPointer();
glIndexPointer();
glTexCoordPointer();
glEdgeFlagPointer();
glGetPointerv();
glDrawElements();
glInterleavedArrays();
```

第 20 章　OpenGL 扩展功能及程序改进

本章主要介绍 Windows 环境下 OpenGL 的扩展功能，以及对从经验中得出的对 OpenGL 程序的绘制性能进行改进的方法。OpenGL 的许多扩展功能都是为改进程序性能设计的，因此首先介绍 Windows 环境下 OpenGL 的扩展功能。

20.1　Win32 环境下 OpenGL 扩展集

OpenGL 扩展集具有一些新的特点和一些增强的性能。OpenGL 扩展集的使用可以使 OpenGL 应用系统的性能得到很大的改善。OpenGL 扩展集中的许多功能是首次出现的，如编译顶点数组扩展可以锁定顶点数组，它可以大大改进系统性能。索引纹理扩展可以在颜色索引方式下进行纹理映射等。

本节从确定扩展集开始，逐步详细介绍 OpenGL 扩展集。本节所要介绍的内容包括以下部分：

- 确定扩展集可用。
- 编译顶点数组扩展集。
- 物体空间顶点拾取扩展集。
- 调色板纹理扩展集。
- BGRA 扩展集。
- 索引纹理扩展集。
- 索引材质扩展集。
- 索引功能扩展集。
- 索引数组格式扩展集。

20.1.1　确定扩展集可用

本节重点介绍如何在系统支持扩展集，首先介绍扩展集的后缀，然后介绍如何检 OpenGL 扩展集是否可用，最后以一个示例说明。

1.　OpenGL扩展集后缀

OpenGL 扩展集的函数带有一个后缀，这个后缀是 EXT 或者是厂家缩写，例如：

```
glVertexPointerEXT()
glColorTableSGI()
```

扩展集内部，如扩展集字符串则使用前缀，如 SGI_color_table。下面是后缀和前缀的

详细说明：

- EXT 已经在多个版本的 OpenGL 扩展集中使用。
- SGI，SGIS，SGIX 在 SGI 公司的不同类型的 OpenGL 版本中使用，带这些字符的某些函数仅在 SGI 工作站中使用。
- WIN 仅在微软公司的扩展集中使用。

2. 检查OpenGL扩展集可用

所有扩展集函数都在 gl.h 中定义，函数的名称可通过调用函数 glGetString()返回的字符串中得到，例如，如果扩展集支持顶点数组扩展，EXT_vertex_array 在 gl.h 中作如下定义：

```
#define SGI_COMPILED_VERTEX_ARRAY 1
```

调用函数 glGetString()；则返回扩展集字符串 GL_SGI_compiled_vertex_array。可在编译时使用在 gl.h 中的定义，从而判断是否在程序中调用了相应的库中所有的扩展函数。

在应用程序中必须进行两个步骤，即编译时检查，例如，确认 GL_SGI_compiled_vertex_array 已经定义；进行运行时检查，例如，确认 GL_SGI_compiled_vertex_array 是函数 glGetString()返回的扩展字符串。

- 编译时检查是检查编译连接环境，主要是.h 文件和库文件。这个步骤保证新函数或新枚举的入口点(entry)已经定义。如果运行环境不支持，则不能编译和连接使用扩展集的程序。
- 运行时的检查是检查运行的环境。这个步骤保证扩展集在运行环境中支持。

注意：*扩展集的是否可用不仅依赖于操作系统，也依赖于程序所运行的环境，即在某些不支持 OpenGL 的显示驱动的环境中，虽然使用了 OpenGL 函数库，但硬件环境却不支持图形的加速。*

至于函数 glGetString()的调用，注意不要在绘制描述表(RC)起作用之前调用。在 X 窗口系统中，还必须知道所显示的机器类型，因为不同的 X 显示环境的工作机制是不同的。在微软 Windows 的环境下，不像 X 窗口系统那样需要知道机器类型，但不能忘记该函数的调用场合。

在当前 OpenGL 绘制上下文中使用扩展集，可调用函数 wglGetProcAddress()获得 OpenGL 扩展集的地址。例如，为使用扩展集函数 glVertexPointerEXT()，在程序中可通过调用函数 wglGetProcAddress()获得该函数的地址，其步骤如下：

(1) 调用函数 wglGetProcAddress()，以获得该函数所在的头文件 gl.h 中的函数指针定义：

```
typedef void (APIENRY *PFNGLVERTEXPOINTEREXTPROC)
(GLint size, GLenum type, glsizei stride,GLsizei count, const GLvoid
        *pointer);
```

(2) 使用从第(1)步中获取的定义声明一个指针，以存储调用函数 wglGetProcAddress()的结果：

```
PFNGLVERTEXPOINTEREXTPROC glVertexPointerEXT;
glVertexPointerEXT = ( PFNGLVERTEXPOINTEREXTPROC )
```

```
wglGetProcAddress ( "glVertexPointerEXT" ) ;
```

(3) 使用第(2)步中的指针执行命令：

```
(*glVertexPointerEXT)(3, GL_FLOAT, 0, 0, data);
```

3. 检查扩展集可用性的示例

检查扩展集是否可用可通过调用函数 QueryExtension()实现，在例 20.1 中就是使用该函数进行扩展集的可用性的检查。

例 20.1 检查 OpenGL 扩展集的可用性

```
main(int argc, char* argv[]) {
...
if (!QueryExtension("GL_EXT_paletted_texture")) {
fprintf(stderr, "paletted texture extension not
supported.\n");
exit(1);
}
...
}
static GLboolean QueryExtension(char *extName)
{
/*
** 在扩展字符串中查找 extName，不能 strstr 函数，因为扩展名可以是其他扩展名
的前缀。虽可 strtok 函数，但因为 glGetstring 函数的返回值是一个常数，也就是
只读的，不满足 strtok 函数对参数的要求。*/
char *p;

extNameLen = strlen(extName);
p = (char *)glGetString(GL_EXTENSIONS);
if (NULL == p) {
return GL_FALSE;
}
end = p + strlen(p);
while (p < end) {
int n = strcspn(p, " ");
if ((extNameLen == n) && (strncmp(extName, p, n) == 0)) {
return GL_TRUE;
}
p += (n + 1);
}
return GL_FALSE;
```

}

20.1.2 编译顶点数组扩展集

在 OpenGL 扩展集中一个重要的提高数据处理性能的方法是编译顶点数组扩展集，即 SGI_compiled_vertex_array。这个扩展集提供了以下的功能：

- 顶点集可以一次性地进行变换然后通过一系列的调用函数 glDrawArrays(), glArrayElement()或 glDrawElements() 使用这些顶点。
- 顶点数组数据可以更快地从内存中传输和重新排列，从而通过软件绘图发生器从缓存区中传至图形适配器，即图形加速卡，或者传送至服务器内存进行间接绘制；
- 由于编译顶点数组定义明确，数组中的数据不能再改变，这样在程序中也可控制这部分数据的使用，也就是说，在程序中可以锁定某部分数据不变，而改变其他没有锁定的数据。

本节首先介绍编译顶点数组扩展集概况，然后详细介绍其使用方法。

1. 编译顶点数组简介

在 OpenGL 新版本中的顶点数组功能是把顶点数据放入数组中，然后使用数组中数据块。通过执行一个命令来构造多个几何实体。而编译顶点数组的方法更有效地使用顶点数组，这种方法可以锁定数组，从而可以多次使用其中的数据。

编译顶点数组的方法可以通过下面一个示例说明，OpenGL 绘制一个网格体是通过绘制由顶点组成的连续三角形序列来实现的，OpenGL 对每个顶点进行一系列的计算，如光照、坐标变换等。如果这些顶点可以共用，则锁定这些顶点的方法效率更高，锁定这些顶点，直至包含这些顶点的所有表面绘制完毕，就像常用的绘制一个圆柱体的示例一样。

圆柱体包括顶面、底面和边上的三角形条带，每个三角形的角同时也是相邻的三角形的角。这样每个顶点需要用多次，因此，可以锁定这些顶点。从而避免在光照和其他涉及顶点计算时进行重复计算。下面的代码段实现这一过程：

```
glInterleavedArrays().....)
glLockArrays().....)
glDrawElements(GL_TRIANGLE_STRIP).....)
glDrawElements(GL_TRIANGLE_STRIP).....)
glUnlockArrays()
```

2. 使用编译顶点数组

实际上，使用编译顶点数组的方法十分简单，实现编译顶点数组仅需要两个步骤：

(1) 调用函数 glLockArraysSGI()锁定当前所使用的顶点数组。

当这些顶点数组被锁定后，OpenGL 对这些顶点进行编译，也就是说，对整个数组数据进行操作，如变换光照或者变换数组数据的结果。

(2) 调用函数 glUnLockArraysSGI()进行解锁。

在函数 glLockArraysSGI()和 glUnlockArraysSGI()之间，必须确保已经按 first 和 count

指定的范围之内的数组单元不再变化。在调用 glLockArraysSGI()和 glUnlockArraysSGI()过程中改变数组的数据，会导致函数 glDrawArrays(),glArrayElement()或 glDrawElements()处于无序的状态。函数 glLockArraysSGI()和 glUnlockArraysSGI()可以包含在显示列表中。如果 glDrawArrays(), glArrayElement()或 glDrawElements()所作用的数组单元超出 first 和 count 所定义的范围，则这些函数的结果将会出现不稳定结果。

另外，如果在 glLockArraysSGI() 和 glUnlockArraysSGI()之间改变状态也会导致性能冲突，因为 OpenGL 必须使顶点重新起作用。

20.1.3 物体空间顶点拾取扩展集

物体空间顶点拾取扩展集定义为 SGI_cull_vertex，使用了一种在物体空间拾取顶点的方法。物体空间顶点拾取比屏幕空间顶点拾取的方法更有效，因为这种方法避免了诸如裁剪和窗口变换等大量的顶点操作。

本节首先介绍顶点空间拾取的工作机制，然后介绍使用顶点拾取的方法。

1. 顶点空间拾取的工作机制

顶点拾取扩展集的工作机制按以下步骤进行：

(1) OpenGL 对每个顶点按视距进行法向量计算，从而确定顶点的 front 面和 back 面.

(2) 如果多边形的所有顶点被拾取，则这个多边形就被拾取。顶点拾取的过程与面的拾取是独立的，物体侧面的多边形可能有两个：front 和 back 面。由于只有当多边形的所有顶点被选取，多边形才被选取。这样可以通过面拾取而不是顶点拾取的方法来正确地拾取具有双侧面的多边形。

OpenGL 可以在所有的变换之前进行顶点的拾取，这样，仅需要少量的变换，这提高的系统的性能。这种在变换之前的拾取可以避开窗口坐标变换，因为物体空间的视点已经定义。

注意：在屏幕空间拾取，OpenGL 需要进行窗口坐标的变换，这对所有顶点的操作都有利，而不仅仅是对网格体。顶点拾取不仅可用于多边形，也可用于其他所有的图形实体，如点和线等。

2. 顶点拾取的使用方法

使用物体空间顶点拾取扩展集需要两个步骤：

(1) 调用 glEnable()，参数为 GL_CULL_VERTEX 开启顶点拾取状态。

当开启 GL_CULL_VERTEX 的状态时，OpenGL 根据顶点的法向量和视点方向向量之间的点积把顶点分为前向顶点后向顶点：

- 如果法向量与视点方向向量的点积小于等于 0。则把顶点定为后向顶点；
- 如果法向量和视点方向向量的点积大于 0，则把顶点定为前向顶点。

通过计算点积确定面的方向，从而拾取顶点，这种方法与通过调用函数 glCullFace()来确定面的方向的方法是相同的。当多边形的所有的顶点被拾取，则该多边形也就被拾取。

用于顶点拾取的视点方向由视点的位置决定，所拾取的视点位置像光源位置一样是同

构的。当位置的 w 分量非零，则视点位置是局部的，每个顶点上的视点方向按从视点位置减去顶点位置计算；当位置的 w 分量为零，则视点位置是非局部的，可以作为所有顶点的视点方向。也就是说，如果当前变换是透视变换，使用局部位置作为视点方向；如果当前变换是正射变换，则使用一个非局部的视点方向。

(2) 指定拾取视点位置。

指定拾取视点位置与投影变换是密切相关的。为了获得正确的结果，拾取视点位置应该与物体空间投影的中心相对应。

指定正确的位置需要技巧，但使用下面的方法可以变得更容易些：

- 在 MODELVIEW 堆栈中进行模型和视口变换。
- 按以下方式指定投影：

  对于透视投影(使用 glFrustum(), gluPerspective())：

  ```
  glCullParameter(GL_CULL_EYE_POSITION_SGI(0, 0, 0, 1));
  ```

  对于正射投影(使用 glOrtho()，near < far)：

  ```
  glCullParameter(GL_CULL_EYE_POSITION_SGI(0, 0, 1, 0));
  ```

  可调用以下函数指定拾取视点位置：

  ```
  glCullParameter*SGI();
  ```

- 当该函数的 pname 参数为 GL_CULL_VERTEX_EYE_POSITION_SGI 时，视点位置在视点空间，并按当前 GL_MODELVIEW 变换的逆变换进行变换。
- 当该函数的 pname 参数为 GL_CULL_VERTEX_OBJECT_POSITION_SGI 时，视点位置在物体空间并可以直接使用。这种情况用于静态屏幕或者 GL_MODELVIEW 变换时不可逆。

下面程序代码段指定一个视点位置，并且开启顶点拾取状态：

```
GLfloat viewPos[4] = {0.0, 0.0, 0.0, 1.0}
glCullParameterfv(GL_CULL_EYE_POSITION_SGI, viewPose);
glEnable(GL_CULL_VERTEX);
```

20.1.4 调色板纹理扩展集

调色板纹理扩展集，即 EXT_paletted_texture，是用于定义一个调色板，以在纹理过滤之前检查纹素(texel)，也就是事先对纹理颜色表进行过滤。

本扩展集定义了新的纹理格式和新的函数，从而在 OpenGL 中支持调色板纹理。

本扩展集提供以下功能：

- 简化压缩纹理图像的方法。
- 对纹理进行调色板的动画。
- 从单个纹理图像中获取不同的图像。

为定义一个调色板纹理，需要一个调色板和一些图像数据，并且图像是按调色板索引的图像。由于调色板纹理需要以上两类信息，这样在定义纹理时增加了工作量。但是通过这种方法可以减少大量的纹理数据，尤其是逻辑视口之外的仍然放在调色板中的冗余信息。

1. 调色板纹理的特点

调色板纹理相对于真彩色纹理而言有以下优点：

- 减少所需要的数据量

使用调色板纹理定义纹理的数据量比真彩色纹理的数据量小得多。假定一个源纹理只有 256 种不同的颜色，大小为 256×256 个像素。真彩色图像的每个像素有 3 个字节，占 192KB。如果把不同的颜色放入调色板中，则每个像素只需要 8 个字节，数据量可以减少到 64KB 加上 768 个字节的调色板数据。如果在纹理中加上 alpha 通道值，则真彩色图像增加 64KB 而纹理图像只增加 256 个字节。这在纹理空间受限制的时候对系统的运行体现得特别重要。对于系统程序而言，使用调色板纹理也可减少内存，从而提供运行性能。

- 容易重复使用数据

调色板纹理使重复使用作为纹理的图像数据更容易。这些纹理大多是类似的，但稍有不同。假定进行驾驶模拟，道路上交通拥挤，有许多轿车相类似，但是颜色图案不同。如果采用真彩色纹理，则每一个颜色图案都需要单独的纹理。如果采用调色板纹理，则对每辆轿车可以使用相同的基本索引数据，对需要的颜色使用不同的调色板即可。

- 可制作调色板特技

调色板纹理可以在调色板显示过程中使用调色板特技。如可以制作简单的动画、过滤、变换以及其他调色板循环效果。所有这些可以在使用有限数据的前提下，使场景的视觉效果更为生动。

2. 使用调色板纹理扩展集的方法

使用调色板纹理扩展集的方法是在调用函数 glTexImage*()时使用下列内部格式参数：

```
COLOR_INDEX1_EXT
COLOR_INDEX2_EXT
COLOR_INDEX4_EXT
COLOR_INDEX8_EXT
COLOR_INDEX12_EXT
COLOR_INDEX16_EXT
```

同时在调用函数 glColorTableEXT()时必须定义调色板。

20.1.5 BGRA 扩展集

BGRA 扩展集，即 EXT_bgra 。这个扩展集使用一个与颜色成分的顺序相反的可选的 BGRA 格式，通过这一格式来扩展主机内存颜色格式的列表。

BGRA 颜色成分顺序与从 DirectDraw 和 GDI 位图中得到的图像相匹配。这个扩展集提供了在 OpenGL 中使用 BGRA 图像的十分简单的方法。

使用 BGRA 扩展集的方法十分简单，即调用函数 glDrawPixels()，glGetTexImage()，glReadPixels()和 glTexImage*()，其中的 format 参数设置为 GL_BGRA_EXT。

下面是一个使用 BGRA 扩展集的代码段：

```
/*
* 在 RGBA 图像，在(10,10)位置画一个 32×32 个象素大小的位图。其中的 image
        是一个指向 32×32 大小的无符号字节型 BGRA 指针的图像。*/
{
unsigned char *image;
glRasterPos2f(10, 10);
glDrawPixels(32, 32, GL_BGRA_EXT, GL_UNSIGNED_BYTE, image);
}
```

20.1.6 索引纹理扩展集

在第 19 章中已经介绍了 OpenGL 1.x 提供的关于纹理环境的新内容，即增加了 3 种新的环境：GL_MODULATE，GL_BLEND 和 GL_ REPLACE。这些新的环境变量使程序可以在没有纹理映射的情况下在纹理图像中改变颜色。

索引纹理扩展集，即 SGI_index_texture ，支持另外 2 种纹理模式，同时，该扩展集支持在颜色索引模式下进行纹理映射。在标准的 OpenGL 中不支持颜色索引模式下的纹理映射。

索引纹理扩展集主要具有以下特点：

- 提供纹理图像的索引内部格式。
- 提供 REPLACE，ADD 和 MODULATE 方式的纹理环境。

使用该扩展集可以指定一个使用颜色索引内部格式的纹理调色板，这个调色板与一个纹理物体联系在一起，纹理调色板只用于纹理图像，而且必须满足下面两个条件之一：

- 纹理调色板有一个颜色索引内部格式，当前的描述表是一个颜色索引绘制描述表。
- 纹理调色板具有一个内部格式成分，当前的描述表是一个 RGBA 绘制描述表。

如果使用了 mipmap 方法，则所有的 mipmap 层必须使用相同的内部格式。

下面是使用索引纹理的程序代码段：

```
GLubyte texture[16] = { 0xff, 0xff, 0x00, 0x00, 0xff, 0xff,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff,};
glTexImage2d(GL_TEXTURE_2D, 0, GL_COLOR_INDEX8_EXT,
4, 4, 0, GL_COLOR_INDEX, GL_UNSIGNED_BYTE, texture);
```

注意：*在这段程序中，参数 GL_COLOR_INDEX8_EXT 用作一个纹理内部格式。另外，如同在 10.1.4“调色板纹理扩展集”一节中所介绍，当纹理内部格式是颜色索引格式时，纹素是按颜色索引像素传送路径装入的。*

当使用扩展集时，扩展集提供一个新的基本内部格式 COLOR_INDEX。这个格式相对于本扩展集提供的 3 个环境变量的作用如下所示：

```
基本内部格式 REPLACE MODULATE DECAL BLEND ADD
COLOR_INDEX Iv = It Iv = If * It undefined undefined Iv = If + It
```

20.1.7　索引材质扩展集

在第 8 章 OpenGL 的关于改变材质属性和光照的内容中，介绍了使用当前 RGB 颜色作为材质的方法，即调用函数 glColorMaterial()，然后开启 GL_COLOR_MATERIAL 状态。本节所要介绍的索引材质扩展集中，则把当前的索引用于颜色索引绘制描述表中的光照计算。使用索引材质扩展集比定义材质的速度要快。

索引材质扩展集，即 SGI_index_material。该扩展集的方法是把一个或多个颜色索引材质属性连接到当前索引表上，从而这些属性可以一直使用索引值。

这种过程的实现十分简单，即在调用函数 glEnable() 或 glDisable()时使用参数 GL_INDEX_MATERIAL，然后调用函数 IndexMaterial 控制索引模式即可。函数 IndexMaterial 的用法如下所示：

```
void IndexMaterial (enum face, enum mode);
```

其中，face 是 FRONT、BACK 或 FRONT_AND_BACK 中的其中之一的参数，表示当前索引所作用的面。

Mode 是 INDEX_OFFSET。

材质属性一旦改变就保留下来，除非给予一个新的索引或者设置一种新的材质。在设置新的材质时，索引材质扩展集处于关闭的状态，这样才能覆盖原来的值。当开启 INDEX_MATERIAL 后，各种参数就与当前的索引连接在一起了。

注意：当 GL_INDEX_MATERIAL_SGI 开启时，像颜色索引模式绘制光照一样，颜色值的计算优先。

在最后一个颜色索引计算完毕后，系统执行以下几个步骤：

(1) 把索引转换成固定位的值，这个值是一个不确定字节数的值。

(2) 选择与固定位最近的值。

(3) 把固定位的值移位 index_shift 个字节，如果这个索引位移 index_shift 大于 0，则左移；反之为右移。

(4) 其他情况下的移位偏移量为 0 。

(5) 把带符号整数偏移量 index_offset 加到索引值。

(6) 使用函数 glMaterial()设置索引位移量 index_shift 和索引偏移量 index_offset，其相应的参数 pname 分别为 INDEX_SHIFT 和 INDEX_OFFSET。这样，颜色索引光照公式也被修改，光照颜色索引可能也移位了，这时，与颜色索引模式下相比，使用光照和雾时需要更多的控制。

20.1.8　索引功能扩展集

索引功能扩展集，即 SGI_index_func，其主要的功能是删除那些不能与一个参考值相比较的面片。

在 OpenGL 的 alpha 检测时，在 RGBA 模式下， alpha 检测可以根据面片的 alpha 值确定保留或删除该面片。索引功能扩展集与此类似，但不是与 alpha 值比较，而是与一

个参考值比较，这种方法把一个面片的索引值与一个参考值相比较。

索引检测在面片操作的剪取检测(scissor test)和 alpha 检测之间进行。使用索引功能扩展集的方法是调用函数：

```
void IndexFuncSGI (GLenum func, GLclampf ref);
```

该函数设置参考值和一个用于检测的比较函数。参考值的范围在 0~1 之间。Func 的值为以下宏之一：

```
GL_NEVER
GL_LESS
GL_EQUAL
GL_LEQUAL
GL_GREATER
GL_NOTEQUAL
GL_GEQUAL
GL_ALWAYS
```

索引扩展集作用的方法是调用函数 glEnable()或 glDisable()设置开启和关闭状态，这 2 个函数的参数为 GL_INDEX_TEST。可以调用函数 glIsEnabled()判断是否开启索引检测状态，glIsEnabled()的 pname 参数为 glGetBooleanv()，glGetIntegerv()，glGetFloatv()和 GetDoublev()。

为获得扩展集的信息，可调用函数 glGet*()，这时该函数的参数为：

INDEX_TEST_FUNC 和 INDEX_TEST_REF

20.1.9 索引数组格式扩展集

OpenGL 1.x 的顶点数组功能可以处理顶点、颜色、颜色索引等数组，可以单独地调用数组、数组集或者数组中的一部分。索引数组格式扩展集，即 SGI_index_array_formats 提供了另外的隔行数组格式，这种格式指定索引颜色而不是成分颜色。

索引数组扩展集的使用方法十分简单，只需要在调用函数 glnterleavedArrays()时，把其中的 format 参数设置为以下参数即可：

```
GL_IUI_V2F
GL_IUI_V3F
GL_IUI_N3F_V2F
GL_IUI_N3F_V3F
GL_T2F_IUI_V2F
GL_T2F_IUI_V3F
GL_T2F_IUI_N3F_V2F
GL_T2F_IUI_N3F_V3F
```

20.2 OpenGL 程序性能的改进

由于 OpenGL 是目前应用十分广泛的图形应用程序界面，有许多硬件厂商研制生产了 OpenGL 加速卡以及驱动程序，因此 OpenGL 程序实际上与程序的应用环境密切相关。如果充分利用硬件性能，可以很大地提高程序的性能，编写高性能的应用程序通常需要更深的理解，尤其是需要为一个专门的应用程序开发特殊的功能、特殊的性能时。本章是在对 OpenGL 深入理解的基础上，为充分利用硬件环境，对改进 OpenGL 程序的方法进行介绍。本章所要介绍的内容如下：

- 改进程序流程。
- 改进几何体子系统。
- 改进栅格化子系统。
- 改进图像流程。
- 快速绘制像素。
- 改进动画性能。
- 进行时间测定。
- 使用 SGI 的 Windows OpenGL 库的一些提示。

20.2.1 改进程序流程

传统的软件改进常常集中在发现和改进重点步骤上，可能需要把整个程序的 90%的时间花在 10%的代码上。程序流程的改进则使用不同的方法，这种方法寻找制约着其他处理过程的瓶颈问题。这在 OpenGL 的实现过程中体现得尤为重要。

所谓瓶颈，就是存在在程序流程中的阶段或层次。提高性能的最好的方法就是在瓶颈处减少时间的消耗。相反，在瓶颈处做的工作越多，或者在程序的其他地方制造新的瓶颈，这样只能更多地降低运行性能。如果不同的硬件支持流程中不同的部分，只要某部分不变成新的瓶颈，则对流程中其他部分的工作量的增加，但性能不会降低。在这种情况下，有时可以在不降低性能的前提下，通过修改应用程序以提高程序的绘图性能。不同的程序会强调流程中不同的部分，因此必须理解在程序的图形流程中产生瓶颈的那些因素。

注意： *在程序的执行过程中，所有的工作是在主 CPU 中进行的。这样，如果栅格化是瓶颈，则在几何流程中增加工作量是没有意义的，因为这样只会增加 CPU 的工作量而减低了性能。*

本节分 3 部分介绍程序流程中的改进，即图形流程中的 3 个阶段模式、找出程序中的瓶颈和影响性能的因素分析。

1. 图形流程中的3个层次模式

图形流程包括 3 个概念性的层次。按照图形的绘制过程而言，所有过程在 CPU 中执行，部分过程在图形加速卡中执行。这里的模式概念在图形流程的任何情况下适用，因为

通过这种概念可以了解时间消耗在什么地方。这 3 个层次是：

- 应用程序

应用程序在 CPU 中运行，对图形子系统发送命令。

- 几何子系统

包括各多边形操作，如坐标变换、光照、生成纹理坐标和裁剪等，这些操作可能有硬件加速。

- 栅格化子系统

各像素的操作，如把颜色值写至帧缓存等简单的操作，深度缓冲区、alpha 融合、纹理映射等复杂的操作，这些可能有硬件加速。在各层次中的工作量的多少依赖于应用要求。例如，假定一个应用程序绘制少量的大多边形。因为只有少量的多边形，在流程中的几何操作的计算量不太大。但这些少量的大多边形在屏幕上绘制的像素却数量很大，因此流程中需要繁重的栅格化工作。为了加快应用程序的绘制速度，必须在栅格化层次中提高速度，提高速度的方法可以是绘制更少的像素，或者按花费时间少的方式绘制像素，如关闭纹理、dpha 融合或深度缓冲区等。另外，如果在每个多边形的操作层次中节省时间，这样就可在不降低性能的前提下增强功能，例如，可以使用一个效果更好的复杂光照模型，或者定义一个更为细致但大小不变的几何体，这样的几何体是由许多多边形组成。

2. 找出应用程序中的瓶颈

找出瓶颈的基本策略是在执行程序过程中计算其所占的时间，然后在不改变其性能的情况下按几种方式修改代码。如果在流程的某一阶段改变工作了量但性能的变化不大，该阶段不存在瓶颈；如果在性能上存在限制的差别，这就找到了瓶颈。这里对瓶颈进行分析。

- 几何体瓶颈

程序在几何体中产生瓶颈称为变换约束(transform limited)。为检测在几何操作中的瓶颈，可以修改程序以在相同速度下绘制相同数量的像素，但几何体的操作减少。例如，在使用光照时，调用参数为 GL_LIGHTING 函数 glDisable()，临时关闭光源。如果程序性能得到提高，则程序中存在几何体瓶颈。这部分内容还要在 20.2.2“改进几何体子系统”一节中介绍。

- 栅格化瓶颈

程序在流程的栅格化层次中产生的瓶颈称为填充速度约束。同样，为检测在栅格化操作，过程中的瓶颈，可缩小物体或缩小窗口以减少激活的像素的数量。如果程序根据物体的大小和窗口的尺寸来修改操作，这样的检测不会有效。另外，也可以通过减少像素操作如关闭深度缓冲区、纹理或 alpha 融合等以减少工作量。如果这些检测中的任一项就可以提高程序速度，则程序在这层次中存在栅格化瓶颈，这部分内容在 20.2.2“改进栅格化子系统”中还要介绍。许多应用程序需要实现各种各样的功能，程序中的各部分着重系统不同的内容 把这样的程序分解成许多小部分就可以把注意力集中于速度最慢的部分。

- 影响性能的因素

在表 20.1 列出了可能限制绘制性能的因素和相应的流程层次：

表20.1 影响性能的因素

性能参数	流程层次
各多边形的数据量	所有层次
程序耗费时间	应用程序
多边形变换速率和模式设置	几何子系统
一帧中的所有多边形的数量	几何体和栅格化子系统
填充的像素数	栅格化子系统
按给定模式设置下的填充速率	栅格化子系统
清屏或清缓冲区的时间	栅格化子系统

20.2.2 改进几何体子系统

本节介绍改进几何体子系统的一系列优化方法，这些方法分以下几部分：

- 使用最优性能的实体。
- 使用顶点数组。
- 使用正确的绘制模式。
- 优化光照性能。
- 高级几何体的改进技术。

1. 使用最优性能的实体

本节介绍如何用最优的实体绘制几何体。按最优方法绘制几何体，主要从以下方面考虑：

- 使用相连接的实体

如连续线段序列、连续三角形序列、三角形扇、连续四边形序列等。相连接的实体不仅减少数据量，而且减少 OpenGL 对每个多边形或每条边界的工作量。

- 使用正规的多边形

这种多边形是凸多边形，而且是平面的，只有 3 个或 4 个顶点。为获得凸多边形，必须在绘制前调用 GLU 实用函数对多边形进行分化(tessellate)，因此显得有些费时。非平面多边形以及包含大量顶点的多边形也是增加了额外工作量。如果在数据库中包含非正规的多边形，再对数据库进行处理，去掉多余的因素，把多边形转换成正规的多边形，然后使用新的数据库进行绘制。在绘制过程中，可以把处理后的多边形放入 OpenGL 的显示列表中。使用相连接的实体可节省额外的资源。

- 对每个顶点的操作减至最少

绘制多边形的速率直接受作用于各顶点的法向量或颜色的影响。对每个顶点设置法向量或颜色而不是调用 glShadeMode()的方法，与只对每个多边形设置一种颜色的方法相比速度更慢，因为需要在额外的数据和重新设置颜色上花费时间。另外，每个多边形的法向量和颜色数量也直接影响包含多边形物体的显示列表的大小。如果模型的后向面可以使用法向量拾取，则即使不使用光照也要对顶点确定法向量。具体可见物体空间顶点拾取扩

展集。

- 把像实体这样的个体组成一个组，尽量减少状态的改变以减少流程的再次改变。
- 使用归一化的法向量。
- 避免不统一的比例操作。

2. 使用顶点数组

顶点数组是十分有效的提高性能的途径。本节对顶点数组的特点加以小结：

- OpenGL 执行过程中可以利用数据格式一致的优点。
- 调用函数 glInterleavedArrays()可以方便地包装顶点数据。包装的顶点数据在 OpenGL 执行中速度明显加快。
- 调用函数 glDrawArrays()可减少其他子函数的调用。
- 调用函数 glDrawElements()不但减少其他子函数的调用，也可减少每个的顶点由于重复使用的计算。其 lock/unlock ，即锁定/解锁功能更加可以减少几何体的计算量。
- 在 glBegin()/glEnd()之间使用顶点数组的不足之处是可能需要更多内存。如果使用 glBegin()/glEnd()，而不是调用 glDrawArrays()或 glDrawElements()，这时尽量把顶点放在 glBegin()/glEnd()之间。

注意：*在顶点公用的情况，为优化性能，要把顶点锁定。具体方法参见 20.1.2 “编译顶点数组扩展集” 一节。*

3. 正确使用绘制模式

本节介绍如何正确使用绘制模式，以减少几何体在性能上的开销。

- 使用快速绘制模式

为了优化性能，可在任何情况下使用平面渲染(flat shading)模式。这样对各实体、各顶点而言减少了光照的计算量，同时也减少了每个实体处理时的各种数据。这对高性能的线绘制尤其重要。

- 有效地使用费时模式

OpenGL 提供了许多功能，这些功能可以产生特殊效果，并且性能优异。尽管如此，这些功能与没有这些功能的场景相比，需要更多的花费。这种情况这里称为费时模式。只有在需要特殊效果、性能和质量时才使用这种模式。

当不需要时不用这些功能。一旦使用了某种功能，当这种效果看不见时几何变换的速率会减慢。例如，即使多边形与显示出来的雾十分接近，或者雾的密度设为 0，使用了雾会减慢多边形的变换速率也在这种情况下，调用函数关闭雾：

```
glDisable(GL_FOG) ;
```

在费时模式下尽量减少变化。在需要变化时，找出最需要的变化。可以考虑下面几个方面：

- 使用最少数量的纹理以避免纹理之间切换时的开销。如果有许多小纹理，就把它们合并成一个单独、有序的大纹理。不要在绘制一个带纹理的多边形时切换一个新的纹理，在大纹理中选择合适的小纹理的纹理坐标。

- 避免改变投影矩阵或改变函数 glDepthRange()的参数。
- 当启动了雾时，避免改变雾的参数。
- 在不同的投影中绘制时关闭雾，在回到正常的投影时再打开。

注意：费时模式的变化，即使这种变化很小，如改变光影模式、深度缓冲区和融合功能。当需要改变状态时，考虑使用函数 glPushAttrib()/glPopAttrib()，这两个函数有时更快。

4. 优化光照性能

OpenGL 提供了许多光照功能，但为实现这些功能而付出的代价是有时需要依赖于硬件。在进行光照设置是注意要做试验。

按一般规则，尽量使用最简单的光照模型；在一个无限的视口中使用一个无限的光源；一些局部效果而言，可以用有限的光源和一个局部的视口替代局部的光源。为使用高性能的光照，使用以下设置：

- 单独的无限光源。
- 设置非局部的视口。在函数 glLightModel()中 GL_LIGHT_MODEL_LOCAL_VIEWER 设置成 GL_FALSE。
- 使用单面光照。在函数 glLightModel()中的 GL_LIGHT_MODEL_TWO_SIDE 设置成 GL_FALSE。
- 关闭 GL_COLOR_MATERIAL 或 GL_INDEX_MATERIAL_SGI。
- 关闭 GL_NORMALIZE。因为这个参数通常在包含一个比例变换的模型视口矩阵变换时才需要，最好进行预处理以减少比例变换。

使用以下方法可获得高性能的光照效果：

- 避免使用局部光源。
- 局部光源比无限光源明显费时。
- 不要频繁地改变材质。

改变材质参数很费时，如果对每帧需要多次改变材质参数，可以考虑重新组织场景以尽可能减少材质的变化，也可考虑使用函数 glColorMaterial()或 glIndexMaterialSGI()自动地改变参数，而不是调用函数 glMaterial()手工改变参数。

下面程序代码说明如何对每个多边形或每个顶点改变环境和漫反射材质参数：

```
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
...
/* 设置环境和漫反射材质参数*/
glColor4f(red, green, blue, alpha);
/* 画三角形*/
glBegin(GL_TRIANGLES);
...
glEnd();
```

- 使用局部视口。把函数 glLightModel()中的 GL_LIGHT_MODEL_LOCAL_VIEWER

设成 GL_TRUE。这时只使用无限光源，这样会降低一点性能。每个额外的局部光源会明显地降低变换速率。
- 双面光照照亮多边形的两个面。这比二次绘制多边形的方法要快得多。当然，使用双面光照比单面光照绘制物体的速度要慢一些。
- 使用归一法向量。
- 尽可能避免比例操作。
- 尽可能避免改变 GL_SHININESS 的材质参数。每次设置一个新的 GL_SHININESS 值需要很大的计算量。
- 在一个 glBegin()/glEnd()的序列中避免使用光照。
- 尽可能在一个 glBegin()/glEnd()的序列中避免调用 glMaterial()。因为这样会引起严重的性能冲突。这种改变材质属性的方法改变颜色，其运行性能会不可预料。这时，使用 glColorMaterial() 或 glIndexMaterialSGI()。

5. 高级几何体改进技术

下面介绍用于改进几何体绘制的更进一步的方法。可以按以下方法绘制复杂表面的物体：

- 使用函数 glAlphaFunc() ，把一个或多个纹理与单个多边形连接在一起，以产生复杂几何体的效果。
- 把纹理映射到一个单独的多边形中绘制一个复杂物体的图像。在物体的图像之外的纹理中把 alpha 值设置为 0 (物体的边界可通过 alpha 值反走样，alpha 值在 0~1 之间)。把多边形的方向面向观察者。为防止在绘制过程中带纹理的多边形的像素 alpha 值为 0，调用函数：

```
glAlphaFunc(GL_NOTEQUAL, 0.0) ;
```

  这种方法经常用于创建像树这样的物体，这样的物体带复杂的边界或有许多空洞透过背景。
- 减少超出视野或太小看不见的物体或多边形。
- 使用雾以增强视觉细节但不绘制细小的背景物体。

20.2.3 改进栅格化子系统

在多边形栅格化单独的像素时需要大量的数据和操作。最典型的操作包括深度比较、Gouraud 光照、颜色融合、逻辑操作、纹理映射以及反走样。本节介绍进行栅格化操作时的改进方法，以提高有限填充应用的性能。本节介绍的方法包括以下几个方面：

- 删除后向面或前向面。
- 改进栅格化子系统的其他考虑。
- 有效使用费时模式。
- 平衡多边形大小与像素的操作。
- 同时清除颜色和深度缓冲区。

1. 删除后向面或前后面

为减少绘制过程中的填充量，可使用删除后向面或前向面的方法。例如，在绘制一个球时，一半多边形在任何时候是后向面。可在变换之后在各面片操作之前进行后向面或前向面的删除。这意味着后向面删除会使得多边形变换变慢，但多边形的填充会更快。在绘制一个包含许多背向多边形的物体时，可以打开后向面删除，在绘制完毕后重新关闭。

任何对大多边形的绘制操作可以变成填充操作。灵活的绘制结构可以减少某些填充操作时的消耗。例如，如果首先绘制背景，则不需要深度缓冲区操作。更好的方法是绘制背景时关闭深度缓冲区，对其他需要深度缓冲区的物体则打开。

在游戏和飞行模拟中经常用这种技术。绘制天空和地面时关闭深度缓冲区，然后可以不需要额外的性能消耗的情况下把多边形绘制在上面。最后，在绘制机场和山脉时打开深度缓冲区。

有许多情况下不需要深度缓冲区。例如，地形、海洋波浪等，这些三维坐标集也是一个高度值数据场，(x,y)表示平面坐标，而 z 表示在这个点上的高度值。可以用很直接的方法绘制这种高度场，即按与视点的距离的大小决定坐标点的先后顺序，然后沿某一边线绘制三角形条带或四边形条。这样倘若不与先前绘制的几何体的任何面片相交叉，则整个高度场的绘制并不需要深度缓冲区检测。除非后来绘制的深度缓冲区几何体可能与高度场交叉，深度值根本不需要写入。在交叉的情况下，高度场的深度值需要写入，但深度检测可通过调用下面函数避免：

```
glDepthFunc(GL_ALWAYS) ;
```

2. 改进栅格化子系统的其他考虑

这些考虑可以分以下几个方面：

- 经过判断，使用 alpha 融合
  alpha 融合是一种费时的操作。最常用的 alpha 融合的应用是透明，这时的 alpha 值表示物体的不透明值。对完全不透明的物体而言，关闭 alpha 融合。
- 避免不需要的面片操作
  对那些不需要面片操作的物体就关闭面片操作，组成绘制流程，在不引起额外的模式切换的情况下，把面片操作减至最小。
- 使用有效的像素格式
- 使用有效的纹理设置
  例如，把 GL_PERSPECTIVE_CORRECTION_HINT 设置成 GL_FASTEST，把纹理环境设置成 GL_REPLACE ，这样常可以很大地改进性能。同样，尽可能避免使用费时的过滤模式(filtering)，如 bilinear filtering 和 trilinear filtering 等。
- 组织绘制过程，把填充操作减至最小
  例如，如果场景中包含巨大的背景多边形，首先在没有深度缓冲区的情况下绘制背景，然后绘制更复杂的深度缓冲区物体。
- 定义纹理时，使用纹理物体
  这样会在绘制过程中减少纹理状态的变化。这部分内容可参见 19.1.4“使用纹理

物体”一节。

3. 平衡多边形大小与像素的操作

多边形大小的优化还可在程序流程中的其他操作中进行，可以分为以下方面：

- 如果多边形太大，而填充速度与其他流程保持一致，则应用整个应用程序的填充速度也受限制。小一些的多边形在流程中可起平衡作用，同时也可加快多边形的填充速度。
- 如果多边形为与其他流程保持一致而太小，则应用的变化受限制。多边形大而少一些，或顶点少一些，这样可在程序流程中平衡，填充速度也可增加。
- 在特别大的多边形填充过程中，尽可能使用最简单的算法，如，背景填充。

4. 同时清除颜色和深度缓冲区

优化栅格化的另一种方法是清除颜色和深度缓冲区，这也是最基本的帧操作。在某些系统中，这普通的方法还能起到很好的性能优化作用。在需要清除颜色和深度缓冲区时，不要单独清除缓冲区，而是调用函数的同时清除：

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

注意： *在此之前，关闭抖动状态。*

20.2.4 提高图像和像素绘制性能

纹理映射是图形绘制中一项十分重要的工作，同时也是一个耗时运算。因此纹理映射性能的提高对于图形绘制性能的提高起着十分重要的作用。而像素在图形绘制中的作用主要是用来绘制最终图像或作为图形绘制的背景，因此对整体性能的影响也非常大。总之，改善图像和像素的绘制性能对于提高图形绘制的整体性能有时可起到事半功倍的作用。

1. 改进图像流程

为在绘制图像或装载纹理时提高性能，可以使用以下方法：

- 按下面程序代码中的方法设置 OpenGL 状态，这样以后的函数 glDrawPixels()或 glCopyPixels()的处理速度会更快。

```
glDisable(GL_ALPHA_TEST);
glDisable(GL_BLEND);
glDisable(GL_DEPTH_TEST);
glDisable(GL_DITHER);
glDisable(GL_FOG);
glDisable(GL_LIGHTING);
glDisable(GL_LOGIC_OP);
glDisable(GL_STENCIL_TEST);
glDisable(GL_TEXTURE_1D);
glDisable(GL_TEXTURE_2D);
```

- 使用字节小的成分，尤其是无符号字节成分的速度快。在每个成分(R、G、B、alpha、亮度和密度)中使用的像素格式长度为 8 个字节。
- 使用更少的成分量，如直接使用 GL_LUMINANCE_ALPHA 或 GL_LUMINANCE 等。
- 找出瓶颈所在。
  与多边形绘制类似，由于传送、处理和栅格化过程中超载，像素的绘制同样存在瓶颈。当所有的模式关闭时，传送路径最有可能受带宽的制约，合理的像素格式和类型的选择将获得很大的好处。这也就是为什么有时使用字节成分的速度快一些的原因。例如，使用包装像素格式 GL_RGB5_A1 ，以按一种 GL_RGB5_A1 的内部格式装载纹理就可加快速度。另外，放大像素可能会产生栅格化瓶颈。
- 对于装入简单图像的情况，可关闭所有像素模式。尽可能关闭深度检测和纹理。
- 一个很大的像素矩阵需要很大的传输量。为了改进图像流程，必须有效地传输像素。这样一个很大的像素矩阵可以分期分批处理，可保持大传输量。这对硬件加速情况下尤其重要，因为图像必须从主机传送至硬件加速卡。

2. 快速绘制像素

为了快速绘制像素，需要设置一些 OpenGL 状态，然后调用函数 glDrawPixels() 或 glCopyPixels()，这样的像素绘制速度会更快。本节通过程序代码段说明。

```
/* 像素操作之前尽量关闭不相关的状态设置 */
glDisable(GL_ALPHA_TEST);
glDisable(GL_BLEND);
glDisable(GL_DEPTH_TEST);
glDisable(GL_DITHER);
glDisable(GL_FOG);
glDisable(GL_LIGHTING);
glDisable(GL_LOGIC_OP);
glDisable(GL_STENCIL_TEST);
glDisable(GL_TEXTURE_1D);
glDisable(GL_TEXTURE_2D);
/* 按像素方式传输 */
glPixelTransferi(GL_MAP_COLOR, GL_FALSE);
glPixelTransferi(GL_RED_SCALE, 1);
glPixelTransferi(GL_RED_BIAS, 0);
glPixelTransferi(GL_GREEN_SCALE, 1);
glPixelTransferi(GL_GREEN_BIAS, 0);
glPixelTransferi(GL_BLUE_SCALE, 1);
glPixelTransferi(GL_BLUE_BIAS, 0);
glPixelTransferi(GL_ALPHA_SCALE, 1);
glPixelTransferi(GL_ALPHA_BIAS, 0);
```

```
/* 关闭减慢 glDrawPixels 的扩展集 */
#if GL_SGI_index_func
(glDisable(GL_INDEX_TEST_SGI);
#endif
```

20.2.5 改进动画性能

本节首先介绍能够加速动画的因素，然后介绍优化帧速率的一些方法。

1. 加快动画速度的因素

一个流畅平滑的动画依靠的是其帧速章。每秒绘制的帧数越多，则动画表现得越光滑。对绘制每帧所需时间的基本影响因素就是前面已列出的表中的因素。

平滑的动画需要双缓冲区。在双缓冲区中，每个帧缓冲区存储当前帧，该帧将通过视频硬件送至显示器，而这时图形硬件在缓冲区中绘制下一帧，但并不显示。当新的一帧可以显示时，系统在缓冲区之间进行切换。在硬件执行过程中，系统必须在屏幕扫描与缓冲区切换之间一直等待，直到下一个垂直跟踪完成为止。这样每次屏幕扫描才能显示稳定的帧，不会出现二帧或多帧。帧速率必须是屏幕刷新时间的整数倍，屏幕刷新时间在 60HZ 的显示器中大约为 16.7 ms。如果每帧的绘制时间比屏幕扫描时间长 n 条扫描线，则系统在缓冲区切换之前等待 n+1 条垂直跟踪线后才继续绘制，这样总共时间为(n+1)*16.7 ms。

总之，在硬件的执行过程中，绘制帧时所费的时间是觉察不到的，除非整个过程需要花屏幕刷新时间的整数倍时间。如果需要观察性能上的提高，必须减少绘制时间，而且比 16.7 ms 栅格扫描时间还少。如果性能上还能忍受，可在不降低性能的前提下增加功能，只要绘制时间不超过当前栅格扫描时间的许多倍。为了有助于显示器时间上的改善，可以关闭双缓冲区。如果不关闭双缓冲区，那很难知道是否接近 16.7 ms 这个界限。

2. 优化帧速率性能

在硬件执行过程中，有助于优化帧速率性能的最重要的方法，也就是在单缓冲区模式下测定时间。在软件执行过程中，在前景缓存中绘制比在背景缓存中绘制的速度要慢。然而，重要的是在时间测定过程中不要包括 SwapBuffers()函数。关于时间测定的内容，将在 10.2.6 节中介绍。另外，可按以下方法优化帧速率的性能：

- 减少绘制时间，该时间控制在屏幕刷新时间的较低倍内。这也是可观察到的性能改善的唯一途径。
- 在 SwapBuffers()函数执行后，进行不带图形操作的计算。
  应用程序在垂直跟踪之间的等待循环过程中，是可以进行任意不带图形操作的计算的。因此，在发送完属于该帧的最后的图形命令后，立即调用函数 SwapBuffers()进行缓冲区的切换。然后进行下一帧的各种计算，执行下一帧的 OpenGL 命令，再调用 SwapBuffers()等，这样一直继续下去。
- 在清屏后进行非绘制的计算。
  清屏费时。如果在清屏后马上进行其他绘制，就需要实施图形流程，使程序延缓。

因此，在清屏后进行非绘制的一些计算。

20.2.6 进行时间测定

为改善程序的性能，计算时间或者确定基准时间(benchmark)是很重要的工作。通过时间的观察，就可了解在程序中各部分在执行过程中所需时间。为获得最接近硬件所能发挥的最优性能，本节介绍一些改进方法，然后进行严格和系统的分析。

1. benchmark 基本常识

benchmark 实际上是系统的一系列时间基准。这些时间基准是通过对系统的详细分析而确定的。分析的主要内容是根据已知的硬件性能特征，检测程序要求系统做什么，计算做这些工作所需时间。把这样计算所得的理想性能与实际观察所得的性能相比较，以继续应用这样的改进技术，一直至二者非常接近。这样，就可得到程序所消耗的确切的时间，从而很有把握改进更好的性能，或者在速度-质量效果之间进行选择。下面列出一些最常用的性能参数：

- 一帧内所有多边形的数量。
- 按给定的多边形类型和模式设置的变换速率。
- 已填充的像素的数量。
- 按给定模式设置的填充速率。
- 清除颜色和深度缓冲区的时间。
- 切换缓冲区的时间。
- 整个应用程序所需时间。
- 变化次数的多少和每次变化所需时间。

2. 获取确切的时间测定值

为获得确切的时间测量值，可以考虑使用以下方法：

- 在一个静态的系统中测定

当测定时间时，系统中不经常操作的变化会占空间。关闭其他的 Windows 应用，例如，在测定时间时不要设置时钟或网络程序。

- 选择的时间测度不要受时间分辨率的限制
- 尽量使用高分辨率的时钟，对测量的时间段至少是时间分辨率的 100 倍。一般原则是 benchmark 的时间至少是 2 秒钟，这样由于读时间时所引起的误差会比整个误差的 100 还少。为测定更快的某种功能，可以写一段循环代码以多次执行测试程序。

注意： 建议在测定时间时尽量使用这样的循环。一定要按简化的方式组织程序。测定时间可调用函数：QueryPerformanceCounter();　QueryPerformanceFrequency();

- 对静态帧制定时间基准

对程序进行修改，以平等地测定每一试验帧的时间。如果场景发生变化，则当前图形流程中的瓶颈也可能发生变化，这样会使时间测定无效。例如，假如测定一架旋转飞机的

绘制时间，可选择单独一帧重复地绘制，而不是让飞机旋转时测定。一旦分析帧的情况后，就按不同的方式突出图形流程，并观察帧，再对帧进行分析和调整。

- 比较多次试验的结果

重复地运行程序，观察在试验中的变化。某些变化可能由于运行其他程序、系统本身的操作和内存的情况等而引起的。

在时间测定的开始和结束读取时间时，调用函数 glFinish()。如果用了硬件加速卡时，调用该函数非常重要。因为在图形子系统中图形命令送至硬件队列中执行，一旦图形流程就绪，图形命令就被执行。这样，CPU 也可立即进行其他计算，包括发送更多的图形命令直至队列被填满为止。

当测定一些图形代码的运行效率时，必须在发送最后一条命令之后，这样就可以避免处理同其他事务的时间。在时间采样之前，在时间测定之后调用 glFinish()。另外，在测试的开始进行时间采样时，也要调用函数 glFinish()，这样就可保证在时间处理之前不会在队列中有剩余的图形命令在时间处理之前。

3. 获取确切的基准结果

为制定一段特殊的代码段的基准性能指标，可使用以下方法：

- 确定有多少多边形正在绘制，并估计在屏幕上大概会覆盖多少像素。
  在数据库中读取多边形时，程序中记下多边形的数量。为确定填充的像素数量，开始时可以作一个感官上的估计。这其中一定要包括被其他面隐藏的表面，并注意是否打开后向面删除状态。更精确的方法是使用反馈模式计算填充的像素数量。
- 确定在所使用的模式设置下系统的变换和填充速率。
- 通过变换速率把多边形的数量分开，这样就可得到每个多边形操作所花的时间。
- 通过填充速率把像素的数量分开，这样可获得每个像素操作所需的时间。
- 测定整个应用程序所需的时间。

为确定在应用程序中执行命令所需的时间，可以按前面找出应用程序中的瓶颈的内容进行试验。整个处理过程主要通过这些方法来完成。在实践过程中，最好通过某些假定立即开始，然后对调整和实验更精确地理解。最后，需要按不同的绘制技术进行实验，并再次进行基准测试，尤其适用出现未料到的情况。

20.2.7 SGI OpenGL for Windows 库使用技巧

目前 Windows 环境下的 OpenGL 库有 Microsoft 公司的 OpenGL 库和 SGI 的 OpenGL 库。从作者的使用体会而言，认为 SGI 的 OpenGL 库的速度更快。SGI 的 Windows OpenGL 进行了许多改进，并且增加了扩展集，版本也在不断地升级。本书中的 OpenGL 库就是 SGI 的升级版本。值得一提的是，SGI 的 OpenGL 库中的一些功能函数往往会添加到以后新的版本中。本节介绍使用 SGI 的 Windows OpenGL 库的一些提示和技巧。

- 使用编译顶点数组扩展集
  这种方法可以锁定与相邻实体公用顶点的物体。这样会减少每个顶点的计算量。

- 使用物体空间顶点拾取扩展集

  这种方法可以改进拾取顶点数据集的性能。扩展集根据顶点法向量拾取顶点，这样可在其他顶点处理之前拾取顶点。
- 在 Windows OpenGL 窗口内使用连续三角形序列和连续四边形序列的速度最快
- 使用纹理替代复杂几何体

  这种方法可以大大地简化几何体。
- 当使用纹理时，考虑使用子纹理功能

  关于子纹理功能部分内容在 19.1.5“使用子纹理”一节中已经详细介绍。子纹理可以使下一级的纹理更有效。调色板纹理则可以进行纹理压缩。
- 仔细考虑选用颜色索引绘制或 RGB 绘制

  在 OpenGL 中，如果只通过软件绘制，则颜色索引绘制的速度更快。例如，颜色索引光照比没有硬件加速的 RGB 光照的速度要快，当然在颜色索引模式下的效果和功能受限。另外，像素格式的字节数少(即 8.位颜色索引)在软件实现上的速度也要快得多。
- 使 RGB 纹理性能最高

  可把纹理环境设置成 GL_REPLACE，把抖动关闭，把纹理过滤器设置成 GL_NEAREST。在这种环境下，RGB 纹理可以与颜色索引纹理的速度一样快。
- 尽量使用隔行数组定义实体

  其中，N3F_V3F 和 T2F_N3F_V3F 二种格式的速度最快。
- 避免单面片操作

  例如，在绘制大背景时首先关掉深度缓冲区，背景绘制完后再打开深度缓冲区。
- 尽量避免使用以下的费时模式

  如，alpha 检测、alpha 融合、模板检验和逻辑操作、雾和反走样、加宽线和点、条(stripple)、多边形偏移(polygon offset)、纹理过滤纹理边界等。

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "MTA0NTkzODEuemlw",
  "filename_decoded": "10459381.zip",
  "filesize": 39708406,
  "md5": "703b63c45ce06c16a7ae44038f3608db",
  "header_md5": "c519401ef2c3ac918872ab27375c406e",
  "sha1": "7d11c0eddf770ecf6aac232dc874efc748753cc0",
  "sha256": "2d9a0f19ce49471dbb5f2ae50ce29030549881aef46cefd17ae24b5a5844c164",
  "crc32": 2409173304,
  "zip_password": "6622Ee",
  "uncompressed_size": 39608352,
  "pdg_dir_name": "10459381_OpenGL \u7a0b\u5e8f\u8bbe\u8ba1_p456",
  "pdg_main_pages_found": 456,
  "pdg_main_pages_max": 456,
  "total_pages": 471,
  "total_pixels": 3349157889,
  "pdf_generation_missing_pages": false
}